

**THINK'S
LIGHTSPEED PASCAL™**

**THINK'S
LIGHTSPEED
PASCAL**

FOR THE MACINTOSH™

THINK

Lightspeed PascalTM

User's Guide and Reference Manual

Version 1, First Edition

THINK
TECHNOLOGIES, INC.

Lightspeed Pascal Software, Copyright © THINK Technologies, Inc. 1986
Lightspeed Pascal User's Guide and Reference Manual, Copyright © THINK Technologies, Inc. 1986
First Printing August 1986
Printed in the United States of America

The Lightspeed Pascal programming environment was developed at THINK Technologies by Peter Maruhnic and John McEnerney, with assistance from John Hueras, Michael Kahl, David Neal and Wynn Newhouse.

The interactive programming environment was conceived by Andrew Singer and Mel Conway, and was first made possible by Terry Lucas, Peter Maruhnic and John Hueras.

The software was tested by Clement Wang, Ted Bailey, Mark Fourman, David Neal, Wynn Newhouse, Steve Lawrence, Diana Bury and Fleet Hill, with assistance from the staff of THINK Technologies.

The manual was created by David Neal, Clement Wang, John McEnerney, Peter Maruhnic and Robert Herold; and by Peter Mui, Sue Willing and Tim O'Reilly of O'Reilly and Associates, Inc., 981 Chestnut Street, Newton, MA.

The project was managed by Robert Herold and Andrew Singer.

"Lightspeed" is a registered trademark of
Lightspeed, Inc., and is used with its permission.

The THINK logo is a service mark of THINK Technologies, Inc.

Finder; System; Imagewriter Driver; ResEdit; RMaker; Macsbug; Macintalk; System Traps, Equates & .Rel Files; and Macintosh Toolbox Interface files are copyrighted programs of Apple Computer, Inc. licensed to THINK Technologies, Inc. to distribute for use only in combination with Lightspeed Pascal. Apple Software shall not be copied onto another diskette (except for archive purposes) or into memory except as part of the execution of Lightspeed Pascal. When Lightspeed Pascal has completed execution, Apple Software shall not be used by any other program.

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESSED OR IMPLIED, REGARDING THE ENCLOSED SOFTWARE PACKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED IN SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

MacLanguage Series is a trademark of TML systems.
Apple and Lisa are registered trademarks of Apple Computer, Inc.
Macintosh is a trademark of McIntosh Laboratory, Inc. and is used by Apple Computer, Inc. with its express permission.

Table of Contents

User's Guide

A Quick Guide to Lightspeed Pascal

Even if You're Planning to Skip the Manual, Please Read This...	P-1
Basic Features of Lightspeed Pascal	P-2
Running a Macintosh Pascal Program	P-3
Running a Lisa Pascal Program	P-4
Creating an Application	P-4

Chapter 1—Introduction

A Compiler That Works Like an Interpreter	1-1
Debugging—Easy and Intuitive	1-2
You've Got it All	1-3
How to Use This Manual	1-4

Chapter 2—Installation and Hardware Requirements

Running With a Single 400K Drive	2-1
Running With Two 400K Drives	2-2
Running With a Single 800K Drive	2-3
Running With a Hard Disk	2-3

Chapter 3—Getting Started

The Project	3-1
Before You Start	3-1
Starting Lightspeed Pascal	3-2
Creating the Bullseye Project	3-2
Adding a File to Bullseye Project	3-3
Opening Program Output Windows	3-4
Running the Bullseye Project	3-4
Re-Running the Bullseye Project	3-6
A Brief Introduction to Debugging Tools	3-6
Viewing the Text of the File	3-7
Stepping Through Bullseye	3-7
Automatically Stepping with Trace	3-8
Stopping at a Specific Statement	3-8

Observing the Value of a Variable	3-10
Making a Run-Time Modification	3-11
Another Sample Program, MiniPaint	3-12
Opening MiniPaint Project	3-13
Running MiniPaint Project	3-13
Resetting the Program	3-14
An Error in MiniPaint	3-15
Fixing the Error—Temporarily and then Permanently	3-16
Saving a Project as an Application	3-17
Leaving Lightspeed Pascal	3-18
Where to Go from Here	3-18

Chapter 4—Editing

Multiple Windows	4-2
Opening an Editing Window	4-2
Closing a File	4-4
Saving Your Changes Without Closing	4-4
Pascal-Specific Features	4-5
A Special Purpose Editor	4-7
Miscellaneous Goodies	4-7
Search and Replace	4-8

Chapter 5—Projects

What's in the Window	5-2
Two Views of the Project	5-3
Compile Options	5-4
More Things to Look At	5-6

Chapter 6—Running a Program

Before Running a Program	6-1
Running a Program	6-1
When Something Goes Wrong	6-2
Compiling, Building or Linking Without Running	6-4
Save Options	6-4
Halting a Program	6-4
Run Options	6-5

Chapter 7—Debugging Your Pascal Program

Debugging in Lightspeed Pascal	7-2
Following the Finger	7-3
Stepping Through a Program	7-5
Tracing a Program	7-5
Stop Signs	7-6
Go versus Go-Go	7-7
Restarting a Halted Program	7-8

The Observe Window	7-8
The Instant Window	7-9
Saving/Reloading Instant & Observe	7-10
Chapter 8—Predefines, Units and Libraries	
Predefined Procedures and Functions	8-1
Using Units	8-2
Standard Libraries	8-4
The EXTERNAL Directive	8-4
Chapter 9—Building Applications and Libraries	
Applications	9-1
Toolbox Initialization	9-2
Lightspeed Pascal Window Initialization	9-2
Libraries	9-3
Calling Libraries via Pascal Interfaces	9-3
Calling Libraries via External Declarations	9-4
Compiler Options in Applications and Libraries	9-5
Compressed Project	9-5
Smart Linking	9-6
Chapter 10—Using Resources	
Using a Separate Resource File	10-1
Combining Resource Files	10-2
An Example	10-3
What's In A Resource File?	10-4
Chapter 11—Interfacing with Assembly Language	
Run-Time Architecture	11-1
Pascal Data Types	11-3
Pascal Calling Conventions	11-5
Interfacing with Assembly Language	11-7
Chapter 12—LightsBug—Debugging at Lightspeed	
Subroutine Call Chain	12-1
Variables	12-3
Registers	12-4
Zones	12-4
Memory Display	12-6
Editing Memory	12-7
A-Traps	12-9
Examining Compiled Code	12-10
An Example	12-10

Chapter 13—Compile Options

How Compile Options Work	13-2
D—Debug Option	13-2
N—Name Option	13-3
V—Overflow Option	13-3
R—Range Option	13-5
The Initialization Directive	13-7
The Asynch Directive	13-8
Using Directives	13-8

Chapter 14—Inside Lightspeed Pascal

Two Worlds	14-1
Bottlenecks	14-4
Exceptions and Traps	14-6

Chapter 15—Menu Reference

The  Menu	15-1
The File Menu	15-2
The Edit Menu	15-4
The Project Menu	15-6
The Run Menu	15-8
The Debug Menu	15-10
The Windows Menu	15-11

Language Reference

Section 1—Tokens and Constants

1.1 Character Set and Special Symbols	3
1.2 Identifiers	4
1.3 Directives	4
1.4 Numbers	4
1.5 Labels	6
1.6 Character-Strings	6
1.7 Constant-Declarations	7
1.8 Comments	7
1.9 Compiler Options	7

Section 2–Blocks, Scope, and Activations

2.1	Definition of a Block	11
2.2	Rules of Scope	12
2.3	Activations	13

Section 3–Types

3.1	Simple-Types	18
3.2	Structured-Types	23
3.3	String-Types	27
3.4	Pointer-Types	28
3.5	Identical and Compatible Types	28
3.6	The Type-Declaration-Part	31

Section 4–Variables

4.1	Variable-Declarations	35
4.2	Variable-References	35
4.3	Qualifiers	36

Section 5–Expressions

5.1	Operators	47
5.2	Function-Calls	54
5.3	Set-Constructors	55
5.4	Type-Casts	56

Section 6–Statements

6.1	Simple-Statements	59
6.2	Structured-Statements	63

Section 7–Procedures and Functions

7.1	Procedure-Declarations	75
7.2	Function-Declarations	78
7.3	Parameters	80

Section 8–Programs

8.1	Program Syntax	89
8.2	Program-Parameters	89
8.3	Unit Syntax	89
8.4	Uses-Clauses	91
8.5	Unit Dependencies	91

Section 9—Input/Output

9.1	Introduction to I/O	96
9.2	Standard Procedures and Functions for All Files	97
9.3	Standard Procedures for Non-Textfiles	101
9.4	Standard Procedures and Functions for Textfiles	102
9.5	Devices on the Macintosh	115

Section 10—Standard Procedures and Functions

10.1	Dynamic Allocation Procedures	119
10.2	Transfer Procedures and Functions	121
10.3	Arithmetic Functions	123
10.4	Ordinal Functions	126
10.5	String Procedures and Functions	127
10.6	Lightspeed Pascal Window Manipulation Procedures	130
10.7	Miscellaneous Procedures and Functions	132

Appendices

Appendix A—ANS Pascal Compatibility	A-1
Appendix B—Porting to Lightspeed Pascal	B-1
Appendix C—QuickDraw	C-1
Appendix D—The Standard Apple Numeric Environment (SANE)	D-1
Appendix E—Interfaces to the Macintosh Toolbox and OS	E-1
Appendix F—Error Messages	F-1
Appendix G—Macsbug Reference	G-1
Appendix H—RMaker Reference	H-1
Appendix I—ResEdit Reference	I-1
Appendix J—Macintosh Character Set	J-1

INDEX

PART ONE

USER'S GUIDE

A Quick Guide to Lightspeed Pascal

Even if You're Planning to Skip the Manual, Please Read This...

This section is a quick summary of Lightspeed Pascal. If you are familiar with the Macintosh and Pascal, you may prefer to experiment with Lightspeed Pascal before reading the manual. However, there are a few things you should know before you start.

- Lightspeed Pascal uses a *Project Document* to organize the files that make up your program. Source files must be added to a project before they can be compiled and run. Source files can be added from disk, or directly from an editing window. The files in the project are displayed in the *Project Window*. See Chapters 4 and 5.
- There are two ways file names are displayed in the Project Window: *by build order*, and *by segment*. Build order lists files in the order in which they will be compiled, from the top down. Segment view shows which files go in which segments. See Chapter 5.
- Options that control Lightspeed Pascal's debugging, error detection, and other features can be turned on or off by clicking on the appropriate letter next to the filename in the Project Window. See Chapters 5 and 7.
- The project keeps track of files that you have edited. When you run your program, only those files you have changed, and any other files that may be affected by the change, are re-compiled. You can control automatic saving of your source files before running. See Chapter 6.
- You can build stand-alone applications of your projects using the **Build & Save As...** command. See Chapter 9.

A quick guide to Lightspeed Pascal's features follows.

Basic Features of Lightspeed Pascal

Here is a list of basic features of the Lightspeed Pascal environment:

<u>Task</u>	<u>What to Do</u>
Start up Lightspeed Pascal	Double-click on the Lightspeed Pascal icon or on an existing project document or source file.
Create a new project	Choose New Project... from the Project Menu.
Open an existing project	Choose Open Project... from the Project Menu.
Add files to the project	Choose Add File... from the Project Menu.
Open a file for editing	If the file is in the Project Window, simply double-click on its name to open it. If not, choose Open... from the File Menu.
Create a new file	Choose New from the File Menu (⌘ N). This will open up an <i>Untitled</i> edit window, which you can save as a file after editing.
Add a new file to the project	Choose Add Window from the Project Menu, when the file is in the front editing window.
Save an edited file	Choose Save from the File Menu.
Search for text	Choose Find What... from the Edit Menu (⌘ W) to set the search string. Choose Find (⌘ F) to actually find it.
Replace text	Choose Find What... from the Edit Menu (⌘ W) to set the search and replace strings. Choose Replace from the Edit Menu (⌘ R) to replace a single occurrence, or Everywhere (⌘ E) to replace all occurrences after the current selection.
Check syntax of a source file	If the file is the front editing window, simply choose Check from the Run Menu (⌘ K).
Build a project	Choose Build from the Run Menu (⌘ B).
Run a project	Choose Go from the Run Menu (⌘ G).
Build an application	Choose Build & Save As... from the Project Menu.
Exit Lightspeed Pascal	Choose Quit from the File Menu (⌘ Q).

Running a Macintosh Pascal Program

Macintosh Pascal programs will run in Lightspeed Pascal with little or no modification. Important differences to note are:

- Lightspeed Pascal doesn't automatically open the text and drawing windows. You will have to open them by hand, or your program will have to explicitly open them with the *ShowText* and *ShowDrawing* calls.
- Lightspeed Pascal doesn't initialize all variables to zero, as in Macintosh Pascal. Your converted Macintosh Pascal program may behave strangely if it depends on this feature.

Appendix B, *Porting to Lightspeed Pascal*, discusses other differences between Lightspeed Pascal and Macintosh Pascal.

To run a Macintosh Pascal program in Lightspeed Pascal, follow these steps:

<u>Task</u>	<u>What to Do</u>
Start up Lightspeed Pascal	Double-click on the Lightspeed Pascal icon.
Create a new project	Choose New Project... from the Project Menu.
Add the program to that project	Choose Add File... from the Project Menu.
Run the project	Choose Go from the Run Menu (⌘ G).

Creating New Programs

<u>Task</u>	<u>What to Do</u>
Start up Lightspeed Pascal	Double-click on the Lightspeed Pascal icon.
Create a new project	Choose New Project... from the Project Menu.
Create a new file	Choose New from the File Menu (⌘ N). This will open up an <i>Untitled</i> edit window, which you can save as a file after editing.
Add the new file to the project	Choose Add Window... from the Project Menu. You'll be asked to save the file if you haven't done so already.
Run the project	Choose Go from the Run Menu (⌘ G). Remember to open any Text or Drawing windows the program might need.

Running a Lisa Pascal Program

Appendix B, *Porting to Lightspeed Pascal*, describes differences between Lisa Pascal and Lightspeed Pascal.

Creating an Application

To build your project into a stand-alone double-clickable application, choose **Build & Save As...** from the **Project** Menu. You must give the application a name, and save it. You can also build your project into a library. See Chapters 9 and 10 for more information.

Chapter 1

Introduction

Lightspeed Pascal is a remarkable product. In order to understand just how remarkable it is, you need to have been programming for a while. If you're new to programming, Lightspeed Pascal will seem natural and obvious—the way things ought to be.

The fact is that until now, there has been nothing like Lightspeed Pascal, except its sister product, LightspeedC, and its predecessor, Macintosh Pascal.

Lightspeed Pascal provides the kind of integrated environment of which most programmers have only dreamed. A fast compiler, an ultra-fast linker, advanced debugging tools, and a full-featured text editor are combined with a unique code management system that makes it possible to turn around a new version of a program in record time.

Developing an elegant, effective program is hard work. It begins with a real-world problem, and an idea for solving it. But an idea, however clever, is only the beginning. The idea must grow into a design. The design must be implemented, and the implementation must be tested. If the program doesn't fully solve the original problem, it must be refined. Even the best programmer rarely gets everything right the first time.

As a result, program development is an iterative, time-consuming process—a cycle of analysis, design, implementation, and testing, repeated over and over again until the program does what its creator set out to do.

Once a programmer has developed a design—a systematic approach to solving the problem—coding the solution in the language of choice generally goes quickly. Most of the programmer's time is spent debugging, revising, and extending the initial version of the program.

The programmer's ability to create, run, debug, and manage multiple versions of a program as quickly and efficiently as possible is a key element in programming productivity. The efficiency and power of the programming environment becomes of paramount importance.

A Compiler That Works Like an Interpreter

Given that most of a programmer's time is spent making incremental changes, a programming environment that makes it easy to update and test a new version of a program is very important.

For this reason, most new programmers start with an *interpreter*, rather than a *compiler*. Because an interpreter remains in memory while the program is run, and because it re-scans a program's source code each time the program is run, it is easier to correlate errors with the source code, and faster to revise and re-run a new version of the program.

However, most serious programming is done with a *compiler*, which translates source code into an independently executable program consisting of machine language or *object code*. Compiled programs run much faster, and don't require the presence of an accompanying support program like an interpreter. Unfortunately, the compilation process itself has in the past tended to be time-consuming and unwieldy—so much so that it encourages poor programming practices.

In theory, one should make only a few small changes to a program under development before testing that the changes work, and that no new problems were introduced by the changes. In practice, however, all of us tend to "batch" changes, because the program build time (the time it takes to edit, compile, link, and launch a new version of the program) is prohibitive. Especially when working with large programs, it is much easier to make a group of changes before recompiling and linking the program. And of course, when one of those changes introduces bugs into other parts of the program, it is that much harder to find out which change caused the problem.

The first remarkable thing about Lightspeed Pascal is that it is blazingly fast - so fast that it will change the way you program. Not only are the compiler and the linker many times faster than any competing programming system, they are part of an integrated package that also includes an editor, debuggers, and automatic "project manager" that keeps track of your edits, and only recompiles source files that have changed since the program was last built.

Program build time is short enough that you will find yourself making a single change, then compiling and running the program to see how it works. As a result, you will find yourself writing better programs. No more coffee breaks while you wait for an application to be rebuilt - you can edit a program, compile, link and run it, and be back in the editor while most compilers and linkers are still grinding away. *In effect, you have a compiler environment that allows interactive program development* - all of the benefits of an interpreter, without any of its drawbacks.

A problem with most compilers is that a conventional compiled language development system has various language objects such as source files, object files, a link-control file, an executable image, administrative files, etc., all of which have to be managed. Each source file that has been changed, or which relies on another file that has been changed, must be recompiled. The object files produced by this recompilation must be relinked with all of the other object files that make up the program. Only then do you have an executable program. It is easy to forget to recompile a changed unit, and as a result to run a version of the program that is "out of phase" with your source code.

Lightspeed Pascal has a simple organizational system. In Lightspeed Pascal, you have only your source files, library files, and a *Project Document*. The Project Document serves as an on-line project administrator for your program development. As mentioned above, Lightspeed Pascal keeps track of whatever changes you make to your source files, automatically compiling and linking wherever necessary, so that all you have to do is edit your program and run.

Debugging - Easy and Intuitive

Where Lightspeed Pascal excels is in the power of its debugging tools. Like LightspeedC, the Lightspeed Pascal compiler automatically flags language syntax errors, and opens an editing window to the offending line in your source file. Lightspeed Pascal also incorporates a powerful source-level debugger. You can:

- Run your program one statement at a time (single-stepping) or in "slow motion" (continuous stepping, or tracing).
- Halt your program at any point during its execution, either by clicking on a "Bug Spray Can" in the menu bar or by inserting "Stop Signs" at specific statements in your code.
- Examine the value of variables and expressions at the point in the program at which it is halted. An "Observe Window" lets you watch the value of variables and expressions in the context of your program. Whenever the program is paused (in single-step mode) or halted, the values in the Observe Window are updated - giving you an inside look at what your program is really doing while it runs.
- Test Pascal statements by typing them in an "Instant Window". They will be evaluated in the context of the program's state at the time it was halted, allowing you to try out different code or patch a bug or fix a runtime error without recompiling at all. In addition to debugging, the Instant Window can improve the quality of the code you write, because you can test statements at any time, without waiting to rebuild your program.

These debugging features were originally made famous by Macintosh Pascal (THINK Technologies' Pascal interpreter for the Macintosh). They are so powerful that even though Macintosh Pascal is an interpreter designed for the educational market, countless Macintosh developers have used it to prototype their applications. Only at the last minute do they port their code to a Pascal compiler for final production.

If you are unfamiliar with what goes on behind the scenes with a compiler, you might not realize just what an achievement Lightspeed Pascal's debugging features are. Interactive debugging with an interpreter is much easier, since the interpreter remains in memory while the program is being run. With a compiler, by contrast, debugging is usually an after-the-fact proposition—in many cases requiring an analysis of the actual machine code generated by the compiler. It is a major advance in compiler technology to have a debugger integrated directly with the compiler in this way. This is especially true because it is a source-level debugger that allows you to examine your program not in machine language but in terms of the higher-level language constructs with which you originally coded it.

And if that weren't enough, you can open a "LightsBug" Window for low-level debugging as well; the experienced programmer can go from high-level language debugging to debugging Macintosh Toolbox calls in the same Lightspeed Pascal environment. You can break at A-traps, view the stack frame, the heap, memory locations, and so on. In addition, Macsbug, Apple's standard assembly language debugger, is supported.

You've Got it All

Of course, integration is the key. The text editor, the compiler, the linker, the debugger—all of the elements of a complete program development system—are integrated into a single, easy to use package.

In focusing on the amazing speed of Lightspeed Pascal's compiler and linker, and its revolutionary (for a compiler) debugging tools, we shouldn't overlook other features of the program, such as its built-in text editor.

As you enter Pascal source code in an editing window, it is automatically formatted or "pretty printed" when you type a semicolon or press Return or Enter. This means that Pascal reserved words such as **program**, **begin**, and **end** are displayed in lowercase boldface type and that certain lines are indented and reformatted in order to make the code easier to read. For example, indentation is used to show the current nesting of structured statements.

Lightspeed Pascal also incrementally checks the syntax of your Pascal program as each line is entered. The syntax is checked as soon as you type a semicolon or press Return or Enter. If an error is found, Lightspeed Pascal displays in an outline typeface the text it believes to be in error.

And of course, all of the features of Lightspeed Pascal are accessible through the use of the traditional Macintosh pull-down menus. At the same time as Lightspeed Pascal is the compiler of choice for advanced programmers, it is also a superb tool for beginners. Anyone who is familiar with the Macintosh and programming fundamentals can begin creating Pascal programs with Lightspeed Pascal the very first time they use it.

Last but not least, Lightspeed Pascal's implementation of Pascal is the most comprehensive language implementation for the Macintosh to date. It is compatible with ANSI Standard Pascal, and all Macintosh Pascal extensions are supported. Existing programs from other Pascals for the Macintosh will execute with little or no modification.

How to Use This Manual

The Lightspeed Pascal documentation covers all facets of Lightspeed Pascal, explaining its functions and philosophy. It is both a guide on using Lightspeed Pascal and a reference for technical information.

Part One, the *User's Guide*, familiarizes you with the Lightspeed Pascal environment and its features.

Part Two, the *Language Reference*, covers Lightspeed Pascal's implementation of Pascal in detail.

Part Three, *Appendices*, contains sections that give tips on porting programs from other versions of Pascal, additional details on support programs from Apple such as Macsbug, ResEdit, and RMaker, details of the Macintosh Toolbox and SANE (the Standard Apple Numeric Environment) Library, a list of the error messages and example of how they might occur, and other useful information that is hard to come by.

We assume that you are already familiar with Pascal and with the Macintosh. As a result, we have tried to avoid covering topics that are well documented elsewhere, focusing only on Lightspeed Pascal and aspects of the Macintosh relating to Lightspeed Pascal.

If you are just learning Pascal, we recommend *Macintosh Pascal*, by Robert Moll and Rachel Folsom (Houghton-Mifflin, 1985) as an excellent introduction to the language. (It is particularly helpful because of the many similarities between Macintosh Pascal and Lightspeed Pascal.) If you are new to the Macintosh, and you don't know how to handle mechanics such as choosing menu items or dragging windows around on the screen, you should refer to your *Macintosh Owner's Manual*.

If you are planning to develop Macintosh applications rather than portable Pascal programs, be sure to have a copy of *Inside Macintosh*, the definitive reference work on the Macintosh (Addison-Wesley, 1986), or Steven Chernicoff's *Macintosh Revealed* (Hayden, 1985).

The *User's Guide* is organized in the order we expect you'll need to read it.

Since Lightspeed Pascal is easy to learn and use, we've included a quick summary of its features as a preface to the manual, so the most eager and confident among you could plunge in without reading the manual, and return to it at leisure.

If you're reading this now, you're already past that point. You should continue reading. After you install the software (Chapter 2), you should read Chapter 3 for a brief walkthrough of the program's main features. If you are very familiar with editing on the Macintosh, you can probably skip Chapter 4, although we have tried to limit the discussion to special features that differ from other Macintosh text editors, so as not to waste your time. You should definitely read Chapter 5, since the Project organization used by Lightspeed Pascal will be new to you unless you have previously used LightspeedC. Chapter 6 describes some of the different ways that you can run your programs, and Chapter 7 describes how to use Lightspeed Pascal's source-level debugging tools.

Chapters 8, 9 and 10 turn from the discussion of the tools Lightspeed Pascal provides, and begin to discuss how to put Lightspeed Pascal to work building Macintosh applications, and how to use Macintosh resources from within Lightspeed Pascal programs.

Chapter 11 discusses how to call assembly language routines from your programs, and Chapter 12 returns to the topic of debugging, this time with a lower-level debugging tool—LightsBug. LightsBug allows you to examine such Macintosh constructs as the stack, the heap, application zones, and so on.

Chapter 13 describes the use of the compiler options, which add code to perform various kinds of checking on your program at runtime. These options are on by default, and you don't need to read this chapter unless you're interested in what effect they actually have.

Chapter 14 gives a more detailed, technical look at various topics connected with Lightspeed Pascal programming. Once you've mastered the basics of how to run the program, you may want to read this chapter for information on how Lightspeed Pascal works its magic.

Chapter 15 gives a brief description of each of the Lightspeed Pascal menu commands, organized in the same order as they appear on the menus. If you get lost, or just want to get a quick idea of what each of the menus provides, you can skip ahead and read this chapter.

The *Language Reference* describes in detail the Pascal implementation that is used by Lightspeed Pascal. You will most likely find yourself referring to it for the finer points of language syntax. The *Language Reference* is not meant to be read through from front to back, but paged through for details on specific topics. It is logically organized, but like all good reference manuals, relies heavily on the index to help you with quick lookup.

The *Appendices* are likewise designed primarily for reference rather than for reading straight through. However, if you are experienced with both the Macintosh and with Pascal, you will probably be most interested in Appendix A, which summarizes the differences between Lightspeed Pascal and other Pascal language products, and Appendix B, which gives tips on porting existing programs.

Appendix C describes QuickDraw (the Macintosh's graphics package), and Appendix D describes SANE (the Standard Apple Numeric Environment), a library for floating point math. Appendix E lists the interfaces for all of the Macintosh Toolbox and Operating System calls that you can make from Lightspeed Pascal.

Appendix F is a guide to the error messages, with examples and explanations. This is sure to come in handy.

Appendices G, H, and I describe the Apple utilities Macsbug, RMaker, and ResEdit, which are shipped with Lightspeed Pascal. Appendix J lists the Macintosh character set.

If you are interested in a specific topic, consult the index in the back of the book.

Chapter 2

Installation and Hardware Requirements

Lightspeed Pascal currently runs on a Macintosh with at least 512K of memory. It can be run with 400K of disk storage, but 800K or more is recommended.

Lightspeed Pascal is shipped on three 400K disks:

- 1) LP1.System, which contains the Lightspeed Pascal application, the *System* (version 2.0), the *Finder* (version 4.1), and the *Imagewriter* (March 6, 1985).
- 2) LP2.Libraries, which contains two libraries, *MacTraps* (register based trap glue and QuickDraw Globals) and *MacPasLib* (Pascal predefines), and two folders. The Libraries folder contains several library object files and their associated interface source files. The BullsEye and MiniPaint Folders contain the demo programs for Chapter 3. The Editor Demo folder contains a text editor demo.
- 3) LP3.Utilities, which contains a .Rel converter for taking assembly language files from the MDS assembler into Lightspeed Pascal format, the Maxbug debugger, and the Apple utilities ResEdit and RMaker.

Before you start, back up all three disks and archive the originals in a safe place. Lightspeed Pascal is not copy protected, so you can make back up copies for your own use. Be sure to only use the backup copies in all your work. Use the originals **ONLY** for making backup copies.

Running With a Single 400K Drive

The strategy here is to create a stripped down generic disk with a MiniFinder and everything you need to use Lightspeed Pascal. You can then use a copy of this generic disk for each new program that you write. You create this disk as follows:

- 1) Copy the LP1.System disk to a new blank disk. Name the new disk *Generic LSP*.
- 2) Reboot your Macintosh using the new disk.
- 3) Select the Lightspeed Pascal icon (click once on it) and then choose **Use Minifinder...** from the **Special** Menu. When the dialog box appears, click the **Install** button to put a *MiniFinder* on your new disk.
- 4) Now throw away the *Finder* (not the newly created *MiniFinder*) by dragging it to the Trash can.

- 5) Reboot your Macintosh using the new disk. The Lightspeed Pascal icon will be visible in the *MiniFinder*. Start up Lightspeed Pascal by double-clicking on the icon.
- 6) Create a new project on the disk by choosing **New Project...** from the **Project** Menu. Name it *Generic Project*.
- 7) Build the project by choosing **Build** from the **Run** Menu.
- 8) Lightspeed Pascal will warn you that the file *MacPasLib* cannot be found, and will ask if you want to find it by hand. Click on the **OK** button or press Return.
- 9) The standard file dialog box is shown. Eject your disk, insert LP2.Libraries and open the *MacPasLib* file.
- 10) After a disk swap, Lightspeed Pascal will ask you if you want to change all subsequent library references in the project to the new disk. Click the **Yes** button or press Return.
- 11) After more disk swapping your project will be built. Close the project by choosing **Close Project** from the **Project** Menu.
- 12) On the new disk you now have a ready-to-use Project to which you can add and build source files. Reboot your Macintosh using the LP1.System disk and make several copies of the *Generic LSP* disk.
- 13) If you plan to use Macsbug, copy the *MaxBug* file from the LP3.Utilities disk to the *Generic LSP* disk copy, and rename it *Macsbug*. Using Macsbug will decrease the amount of disk space left for your programs.
- 14) To start using Lightspeed Pascal, reboot your Macintosh using one of the copies of *Generic LSP*. Without Macsbug on the disk, you should be able to write programs of up to about 1000 lines before the disk is full. If you remove the Imagewriter (making it impossible for you to print programs or screen shots) you can get up to around 2000 lines.

Running With Two 400K Drives

Boot your backup copy of LP1.System and put the backup of LP2.Libraries into the other drive. The LP1.System disk should always remain mounted.

You can create new projects on the LP2.Libraries disk. New projects by default contain the MacTraps and MacPasLib libraries, and Lightspeed Pascal will attempt to find these files on the boot volume (LP1.System). Since they are on the LP2.Libraries disk, you will be asked to open them by hand whenever a project is built for the first time. When asked if you want to change all references of LP1.System to LP2.Libraries, choose yes, and you will have no further problems.

You also can create projects on a new blank disk. The first time a new project is built, you will have to open *MacTraps* and *MacPasLib* by hand as described above.

You can avoid this problem by copying the *MacTraps* and *MacPasLib* libraries to the LP1.System disk. You must first make room for them by installing the MiniFinder and removing the Finder as described above for single drive users.

Alternatively, you can use the same trick as described for single drive users. Create a project on a blank disk, build it to get *MacTraps* and *MacPasLib* loaded into the project, then just make copies of this project rather than keeping the libraries around.

Running With a Single 800K Drive

Copy the contents of LP1.System and LP2.Libraries to a blank 800K disk. Re-boot with the new disk. There should be plenty of room to do whatever you want. If you run out of space, you can remove unneeded library files from the disk. They can of course be moved back to your working disk whenever you write a program that needs them.

Running With a Hard Disk

Copy the Lightspeed Pascal application from the LP1.System disk to your hard disk. Copy the *MacTraps* and *MacPasLib* libraries and the Libraries folder from the LP2.Libraries disk to your hard disk. Put them in the same folder as Lightspeed Pascal.

You can create projects wherever you want on your hard disk.

Lightspeed Pascal is compatible with Apple's Hierarchical File System (HFS). However, if you are using HFS, you should not use the System & Finder that are provided on the Pascal.System disk. Keep the System & Finder that you are currently using. Preferably, use System 3.2 and Finder 5.3, which have just recently been released by Apple. There are known bugs in System 3.0, 3.1 and 3.1.1, although Lightspeed Pascal has built-in workarounds so that you can use these System versions if there is no alternative.

Chapter 3

Getting Started

This chapter provides a discussion and walk-through of many of Lightspeed Pascal's features. It introduces the concept of a *project*, and describes how to create a new project, add files to it, compile the files, and run the project. The chapter ends with an introduction to Lightspeed Pascal's source-level debugging tools.

The Project

Lightspeed Pascal has a simple organizational system. All of the files associated with a particular program under development are organized into a *project*. Instead of having to keep track of object files and numerous other miscellaneous files normally associated with development systems, in Lightspeed Pascal you have only your source files and a *project document*.

The project document is an on-line project administrator for your program development. It contains the information needed to compile, link, and run your program:

1. A list of all the files that comprise your total program.
2. The *object code* that results from compiling each source file.
3. Information on whether or not a file has been edited since it was last saved.
4. Information on any object code *libraries* used in the project.

Information about the currently active project is displayed in the *Project Window*.

When you want to run your program, Lightspeed Pascal first *builds* your project. This involves loading any libraries needed by your program, compiling all the files that have been changed since the last build, and linking the object code produced by the compilation. If, after running, you do not make any changes to the project or to files in the project, these steps are not repeated when re-running your program.

We'll walk through all these steps in this chapter. Complete information on the project document is given in Chapter 5, *Projects*.

Before You Start

This chapter shows how to create a new project, using demonstration files included with Lightspeed Pascal. This demonstration is not rigorously designed as an on-screen tutorial, but you should be able to follow it on your system as well as on paper.

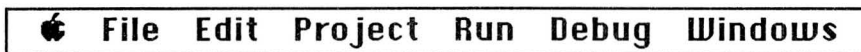
This walk-through assumes that your system is configured with two 400K disk drives, and that copies of the Lightspeed Pascal distribution disks LP1.System and LP2.Libraries are in those

two drives. If your system is not configured this way, you may have to adjust the sequence of steps given here to locate files. However, you can still follow this walk-through if the necessary files can be made accessible to your Macintosh's storage system. Lightspeed Pascal can be found on the LP1.System disk, and the demonstration files can be found on LP2.Libraries. If you have already used your disks, the file organization may not match the illustrations shown in this chapter.

See Chapter 2 for information on how to install Lightspeed Pascal.

Starting Lightspeed Pascal

Double-clicking on the Lightspeed Pascal icon will start up the Lightspeed Pascal development environment. A menu bar like this will appear at the top of an otherwise blank screen:



Our first program will draw a Bullseye on the screen. In order to get to that point, we will create a project, add the file *Bullseye* to the project, open the Drawing Window, and run the project. Let's start at the beginning.

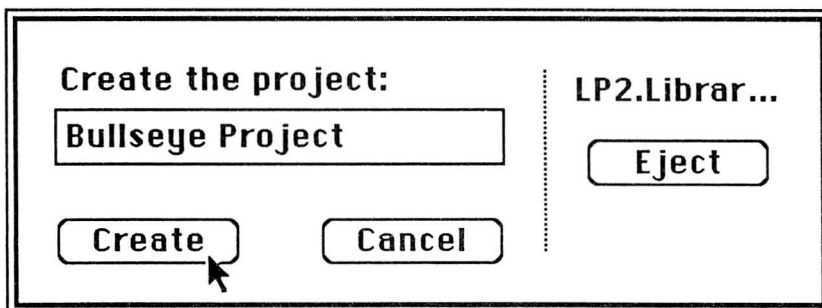
Creating the Bullseye Project

Programs must be added to a project before they can be executed. Choose **New Project...** from the **Project** Menu.

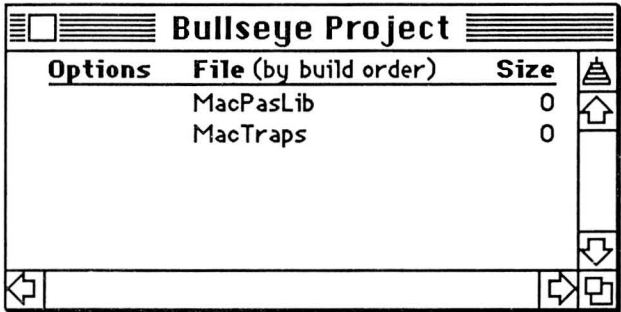
You should create the project on a disk with enough space. The default disk is LP1.System; however, there is not enough room to create a project on that disk. Click the **Drive** button in the **New Project** dialog box to select LP2.Libraries as the disk on which to create the project.

You will be asked to name the project. Let's call it *Bullseye Project*.

Click the **Create** button or press Return to create the project.



A *Project Window* will open on the right side of the screen.

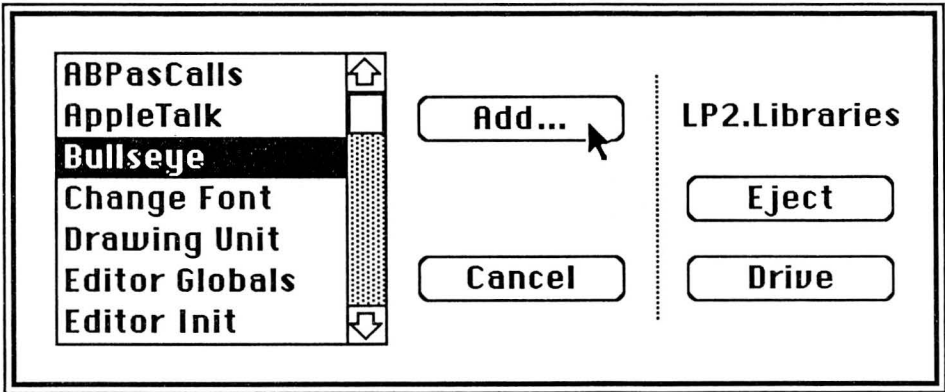


The name of the project, *Bullseye Project*, appears as the title of the Project Window. Inside the window, there are three columns: **Options**, **File**, and **Size**. We won't stop to discuss what these are right now. (Chapter 5 gives details on the contents of the Project Window.)

In addition, the Project Window contains two file names, *MacPasLib* and *MacTraps*. These files are libraries that are included automatically in each project. *MacPasLib* contains Pascal predefined routines, and *MacTraps* contains routines for accessing portions of the Macintosh Toolbox.

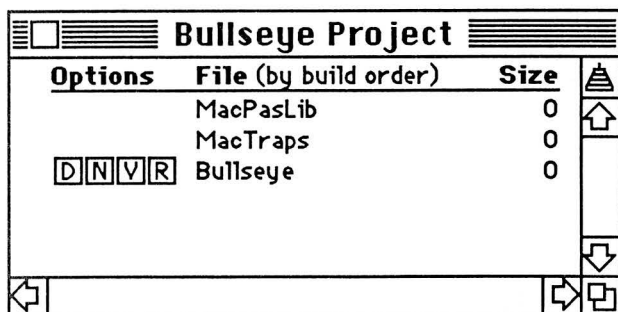
Adding a File to Bullseye Project

Choose **Add File...** from the **Project** Menu to display the files that can be added to the project. Select *Bullseye* from the LP2.Libraries disk, and click the **Add...** button to add it to the project.



Once the file has been added, the dialog box will reappear, allowing you to select another file. Since this project has only one file, don't add any other files. Click the **Cancel** button to make the dialog box go away.

Your Project Window should now look like this:



The name of the file, *Bullseye*, has appeared in the Project Window under the column titled **File**. Notice that the **Size** column reads 0. This tells you that the source file has been added to the project, but has not been compiled. Likewise, the size of the two libraries is listed as 0, since they have not yet been loaded.

Because this project contains only a few files, the Project Window doesn't need to be as large as it is. Make it smaller by using the size box, so that it looks like the picture shown earlier.

Opening Program Output Windows

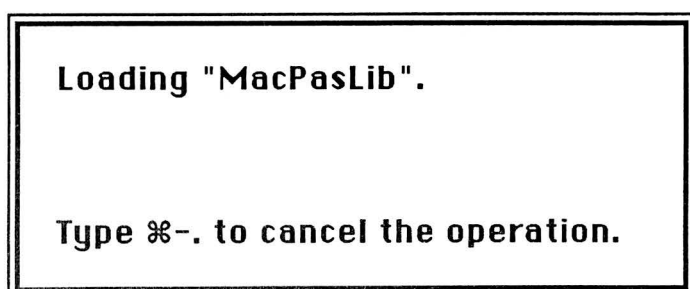
If you've been using Macintosh Pascal, you're probably accustomed to having the Text and Drawing Windows open automatically when your program runs. In contrast, Lightspeed Pascal's pre-defined Text and Drawing Windows have to be opened explicitly. All output from *writeln* statements in your program will go to the Text Window. All drawing output will go to the Drawing Window.

The Bullseye program uses the Drawing Window, so choose **Drawing** from the **Windows** Menu. The Drawing Window will appear.

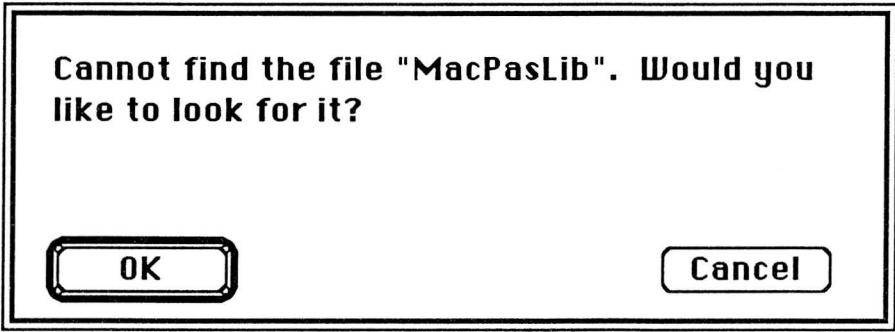
Later on, you'll learn how to open the Text and Drawing Windows automatically, but this sample program requires that you to open them manually.

Running the Bullseye Project

To run *Bullseye Project*, choose **Go** from the **Run** Menu. Several boxes will appear. First, a box will indicate that the *MacPasLib* library is being loaded.

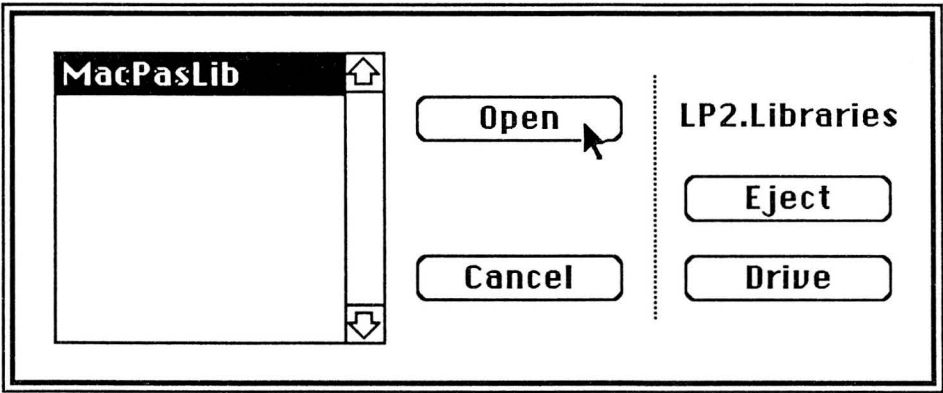


Next, you'll see the following dialog box.

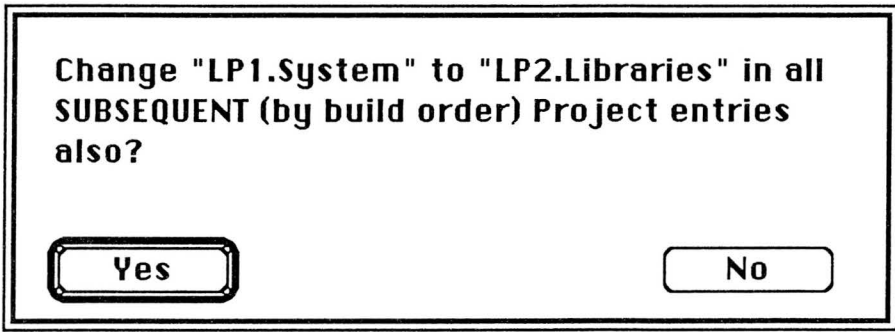


Lightspeed Pascal expects to find MacPasLib on the same disk as Lightspeed Pascal itself. However, it is not distributed this way due to disk space limitations.

MacPasLib is on the LP2.Libraries disk. To find the library, click the **OK** button to open a dialog box that lets you look for the *MacPasLib* library.

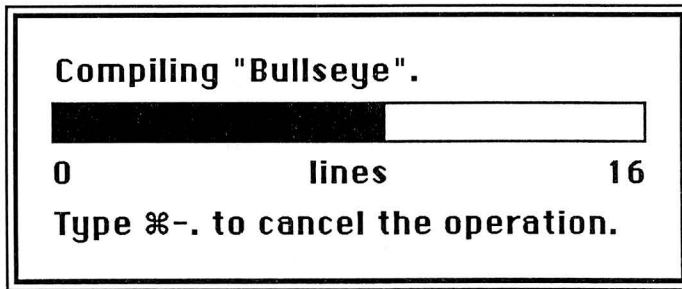


Select the *MacPasLib* library, and click the **Open** button. A second box will appear, asking you whether or not to change all subsequent references from the LP1.System disk to the LP2.Libraries disk.



We do want to change all the references, so click the **Yes** button.

The *Bullseye* program now loads and compiles. You will see a box with a "speedometer" in it while this is happening.



After a brief pause (for disk access), you will see the Lightspeed Pascal compiler flash through this short program. A box will appear, informing you that the project is being updated. Notice that the **Size** column in the Project Window now shows the number of bytes of object code produced.

The program runs, and a bullseye appears in the Drawing Window. You may also notice that a Bug Spray Can appears briefly in the upper right corner of the screen. The Bug Spray Can appears whenever the program is running, and it goes away once the program is done. (Clicking the Bug Spray Can halts the program for debugging. Its use is described in Chapter 7.)

Re-Running the Bullseye Project

Choose **Go** from the **Run** Menu again. This time, no compilation windows should appear, and *Bullseye* will run again. The Bug Spray Can appears in the upper right, and the Bullseye will be drawn in the Drawing Window.

Note that the compilation steps weren't repeated. The project document keeps track of changes to files in the project. Since the Bullseye program wasn't edited, it didn't need to be re-compiled.

A Brief Introduction to Debugging Tools

When a program works just the way you want it to, you are happy just to see it run. But if it isn't working, or isn't working properly, you want to be able to take a closer look.

Lightspeed Pascal's integrated environment provides a number of tools that allow you to look "inside" a program as it executes. You can open the file containing your source code in an editing window, then step through your program a statement at a time. An *Execution Finger* points to each statement as it is executed.

You can also "trace" your program—running it in "slow motion" rather than one statement at a time—or you can insert "Stop Signs" to halt it at any point you choose.

Once your program is halted, you can use Lightspeed Pascal's remarkable Observe Window to watch the values of variables or expressions, or can even change variable values—without recompiling—using the Instant Window.

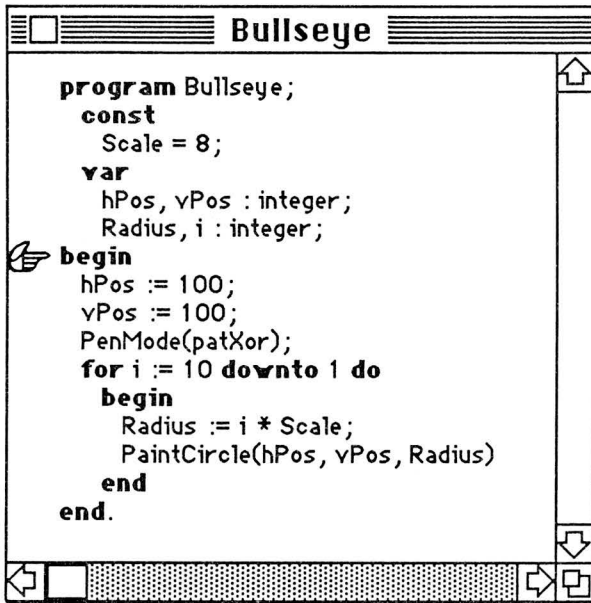
Even though the Bullseye program does work, let's take a moment to see how these tools are used.

Viewing the Text of the File

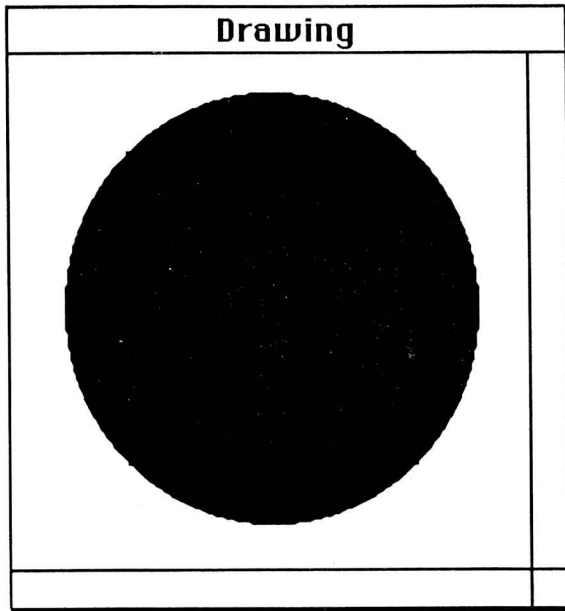
What does the actual text of *Bullseye* look like? Select the Project Window by clicking anywhere inside it. Double-click the name "Bullseye" in the Project Window. A *Pascal Editing Window* appears with the text of *Bullseye*. However, this window is so large that it partially hides the Project and Drawing Windows. Shrink the editing window so that the other windows are completely visible.

Stepping Through Bullseye

Often, you will want to step through your program a statement at a time. Choose **Step** from the **Run Menu**. The Execution Finger, which points at the next statement to be executed, appears next to the **begin** statement in Bullseye.



Choose **Step** again and observe how the finger moves through the file, statement by statement. Continue to step until the finger is pointing at the line starting with *PaintCircle*. This is the line that draws the circles. Step again, and the outermost circle of the Bullseye appears in the Drawing Window.

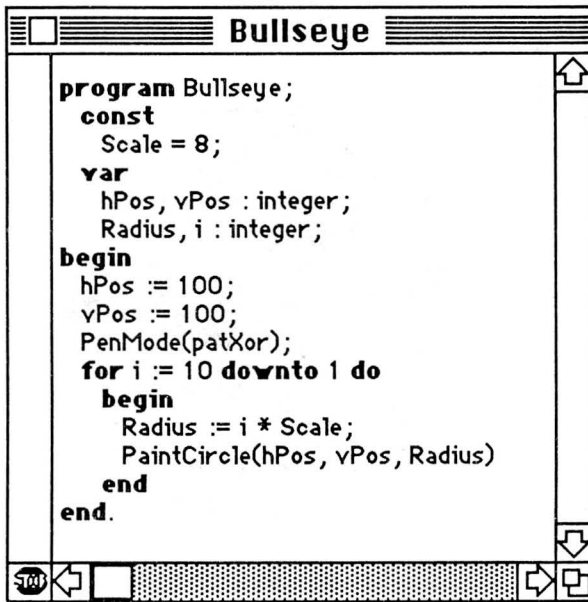


Automatically Stepping with Trace

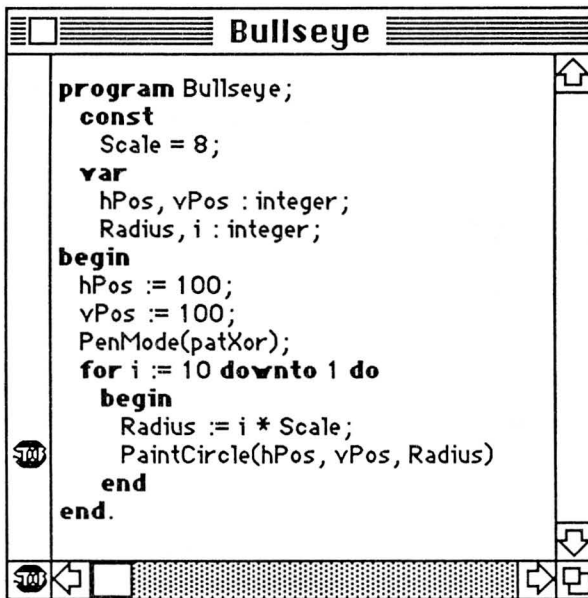
Choose **Trace** from the **Run** Menu. With **Trace**, the program executes faster than with **Step**, but slow enough for you to get an idea of what's happening. *Bullseye* will finish all of its steps in a few seconds.

Stopping at a Specific Statement

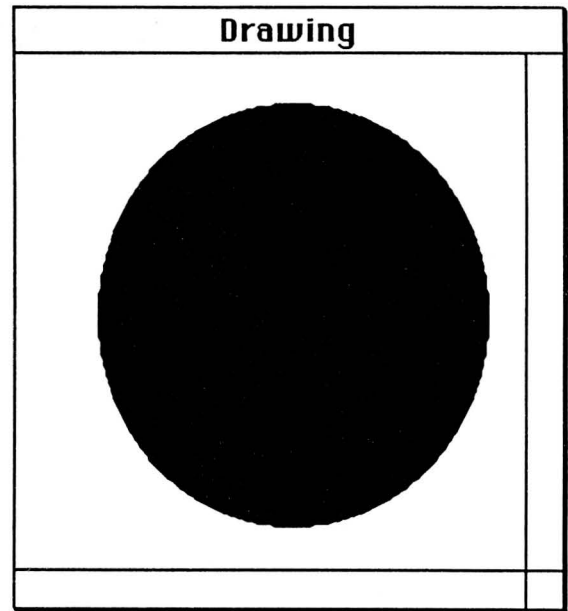
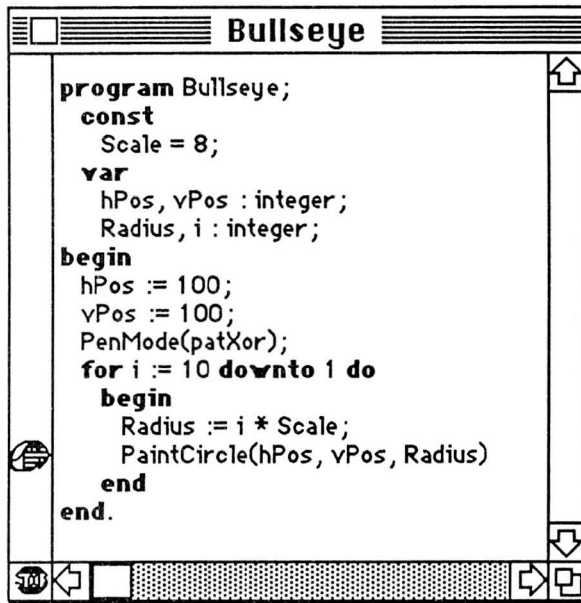
Suppose you wanted to stop at a specific statement in the program? You could **Step** all the way to that point, but that might take a long time (above, it took seven steps to get to the statement with *PaintCircle* on it). The best way is to put a *Stop Sign* in front of the line. Choose **Stops In** from the **Debug** Menu. The editing window will change to have a Stop Sign in the lower left-hand corner, and a vertical bar along the left side.



When you move the pointer into the bar on the left side of the editing window, it becomes a Stop Sign. Move the Stop Sign pointer until it is directly to the left of the statement starting with *PaintCircle*, and click the mouse. A Stop Sign will be left behind at that point.



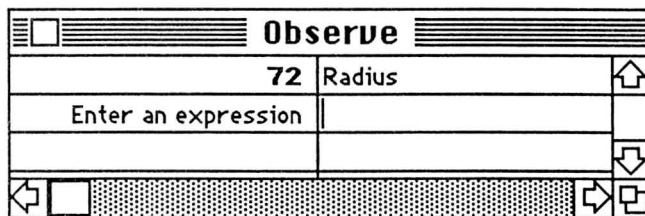
Now choose **Go** from the **Run Menu**. Execution proceeds until the statement with the Stop Sign is encountered, and halts there. The finger points at *PaintCircle*, and covers the Stop Sign. Choose **Go** again, and the outermost circle will appear in the Drawing Window, and execution will again halt at *PaintCircle*.



Observing the Value of a Variable

The size of the circle drawn by *PaintCircle* changes with the value of the variable *Radius*. You can watch this value change with the Observe Window.

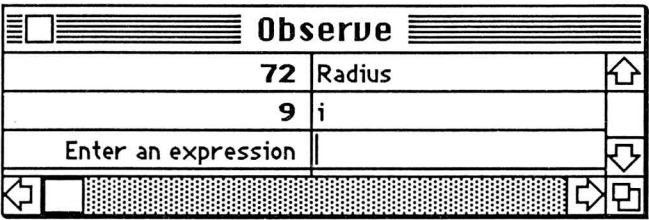
Choose **Observe** from the **Windows Menu**. Type *Radius* in the cell to the right of the line on which it says "Enter an expression", and press Enter. The current value of *Radius* will be displayed in the cell to the left.



If you see the message "no context" in the left cell, your program probably ran to completion because you missed putting in a Stop Sign. Lightspeed Pascal can show a variable's value only when the program is running—that is, when there is a *context* where the value can be found. Place the Stop Sign (see previous page), and run the program again using **Go** so that it halts at *PaintCircle*. The new value of *Radius* will be displayed in the left-hand cell.

A Stop Sign will also halt a program which is run with **Trace** command. Choose **Trace**, and watch the behavior of Lightspeed Pascal and the program. The program will continue from the Stop Sign, and trace through the statements in the drawing loop until it reaches the Stop Sign again. Execution will halt at that point, just as it did when it was started with **Go**.

More than one expression can be put in the Observe Window. For example, you can watch the values of both *Radius* and *i* simultaneously by selecting the Observe Window and typing *i* in the cell underneath *radius*.



You can watch the value of the variables and expressions typed into the Observe Window change dynamically by running the program with the **Go-Go** command. The program stops at the Stop Sign just long enough to update the Observe Window. (If there were no Stop Signs in the editing window, **Go-Go** would act just like **Go**.)

Choose **Go-Go** from the **Run** Menu, and watch the values of *Radius* and *i* in the Observe Window.

Experiment by running the *Bullseye Project* with **Go**, **Go-Go**, **Step**, and **Trace**, and with Stop Signs in different places. To temporarily hide the Stop Signs, uncheck the **Stops In** command in the **Debug** menu. To remove a Stop Sign, click on it. To remove all the Stop Signs at once, choose **Pull Stops** from the **Debug** Menu.

Making a Run-Time Modification

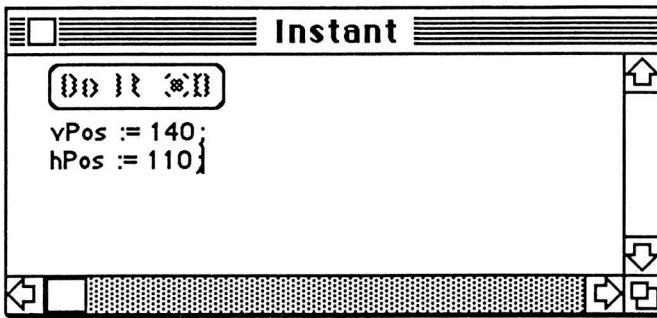
Suppose you are in the middle of running a program, and you want to change the way it executes. In almost any other compiler development system, the process would involve many steps. You would have to stop your executing program, edit the source code, re-compile it, link it, and run it again, repeating this process until the program did just what you wanted.

Lightspeed Pascal's Instant Window allows you to change variables or execute any valid Pascal statement while the program is halted. This allows you to recover from runtime errors or just to experiment with different approaches to solving a programming problem.

Suppose that, after having drawn one or two circles, you wanted the remaining circles to be drawn with their origin at a new position. The variables *vPos* and *hPos* in the program specify the position of the origin. We can change the values of these two variables by assigning them new values in the Instant Window.

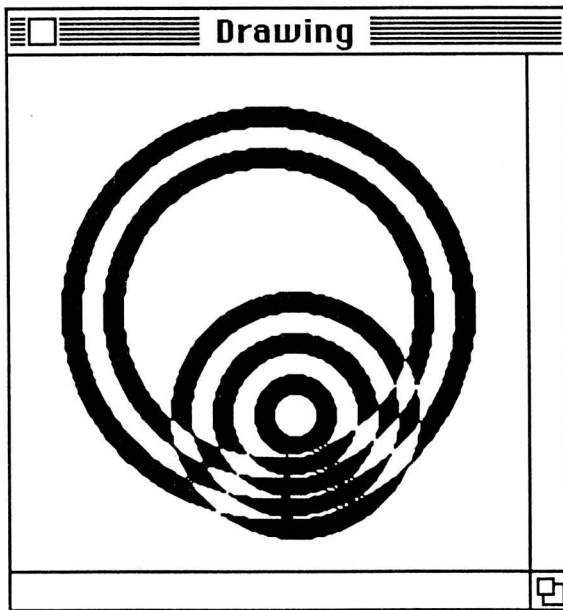
While the program is halted, choose **Instant** from the **Windows** Menu. Move the Instant Window so that it covers as little of the Drawing Window as possible.

Change the value of *vPos* from 100 to 140, and *hPos* from 100 to 110, using valid Pascal syntax as shown in the example below. Click the **Do It** button to execute the change.



Start the program running again. Voila! The program will draw circles centered at the new location.

Assuming that you did this after four circles have been drawn (*PaintCircle* has been called four times), the Drawing Window should look like this:



Although we've demonstrated these tools with a properly working program, they are really most useful when you're debugging an improperly working program. Using the Execution Finger, Stop Signs, and the Instant and Observe Windows, you can watch your program in operation. This allows you to determine exactly what has gone wrong.

To continue on with this chapter, we're going to introduce another sample program. First, close *Bullseye Project* by choosing **Close Project** from the **Project Menu**.

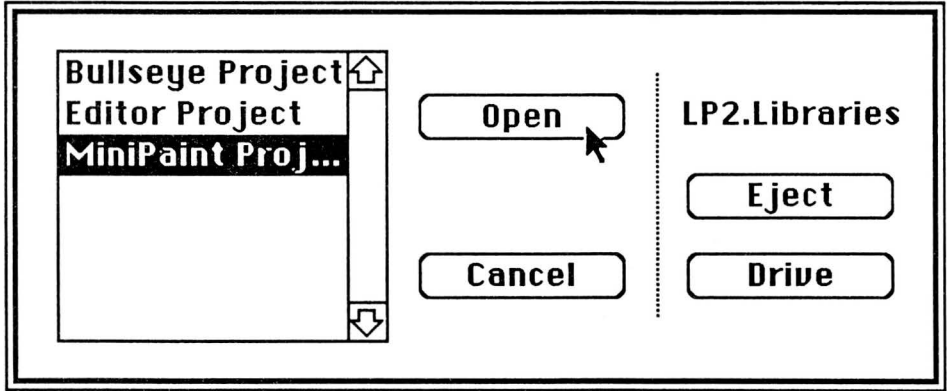
Another Sample Program, MiniPaint

MiniPaint is a program which allows you to draw in the Drawing Window, using the mouse as your "pencil". It should look suspiciously similar to the popular Macintosh program, *MacPaint*. To further demonstrate debugging, we've introduced an intentional runtime error in the Invert

command. In this section, we'll build the project, then fix the runtime error with the Instant Window.

Opening MiniPaint Project

Choose **Open Project...** from the **Project** Menu. Select *MiniPaint Project* on the LP2.Libraries disk, and click on the **Open** button.



A Project Window appears, with the names *MacPasLib*, *MacTraps*, *MiniPaint Globals*, *Drawing Unit*, *Init Unit*, etc. in the window. You may want to adjust the size of the Project Window using the size box.

MiniPaint resides on the disk as several separate files. There is the main program, *MiniPaint Main*, and several associated units which perform specialized tasks: *MiniPaint Globals*, *Drawing Unit*, and *Init Unit*.

<i>MiniPaint Main</i>	contains the "main loop" for the program.
<i>MiniPaint Globals</i>	defines many of the the global constants, types, and variables used in the program.
<i>Drawing Unit</i>	contains routines that handle all of the actual drawing.
<i>Init Unit</i>	initializes the palette, flags, and global values used by MiniPaint.

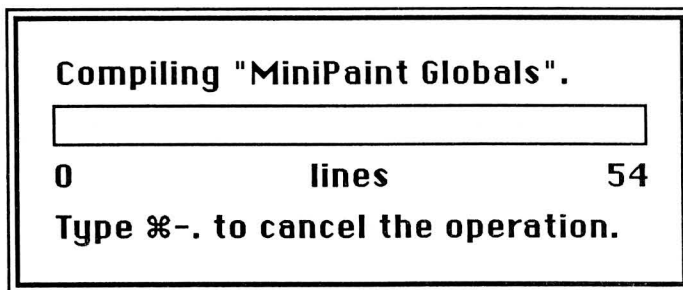
Lightspeed Pascal encourages this sort of modularity, since it takes care of most of the housekeeping normally associated with breaking up a program into multiple files. There are significant advantages to doing this. Smaller files compile faster. They are also easier to debug since problems are more easily localized.

Running MiniPaint Project

Choose **Go** from the **Run** Menu.

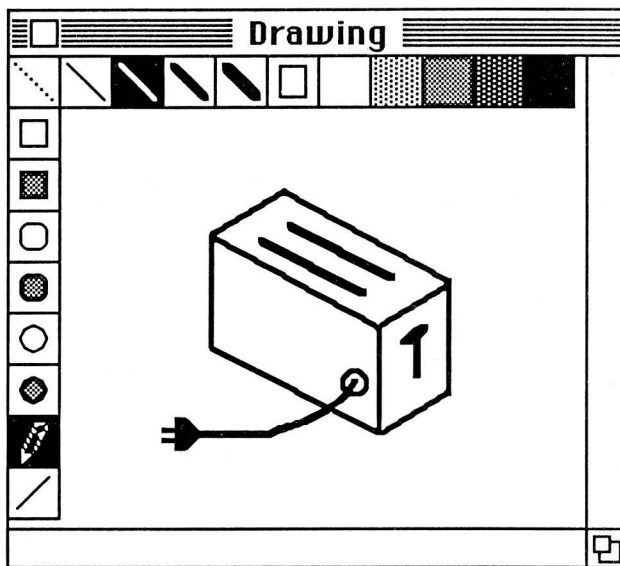
The two default Pascal Libraries will load, as with the first demonstration program. Compilation

boxes will then appear for each unit of the project. First, a window will appear showing the compilation being performed on *MiniPaint Globals*.



Similar boxes will follow for *Init Unit*, *Screen Unit*, and the other files in the project.

The linking step occurs quickly and invisibly. Almost immediately after the last of the compilation boxes disappears, *MiniPaint* will start to execute. To draw, move the mouse into the Drawing Window, and hold down the button to draw. Click on the items in the palette to select them, just as you do in MacPaint. Feel free to play with the program—but don't use the **Invert** command in the **Paint** Menu just yet, since we've intentionally introduced a bug into that part of the program.



Note that the program automatically opens a Drawing Window. It does this by including a *ShowDrawing* call before it actually draws the palette. (An analogous routine, *ShowText*, can be used to open the Text Window.) See Section 10 of the *Language Reference* for more information on *ShowDrawing* and *ShowText*.

Resetting the Program

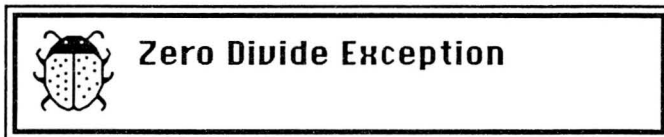
Halt execution of MiniPaint by clicking on the Bug Spray Can in the upper right corner of the screen. The Project Window will appear, with the Execution Finger pointing at *Minipaint Main*.

This shows that at the time you hit the Bug Spray Can and halted your program, a statement in the *MiniPaint Main* file was being executed.

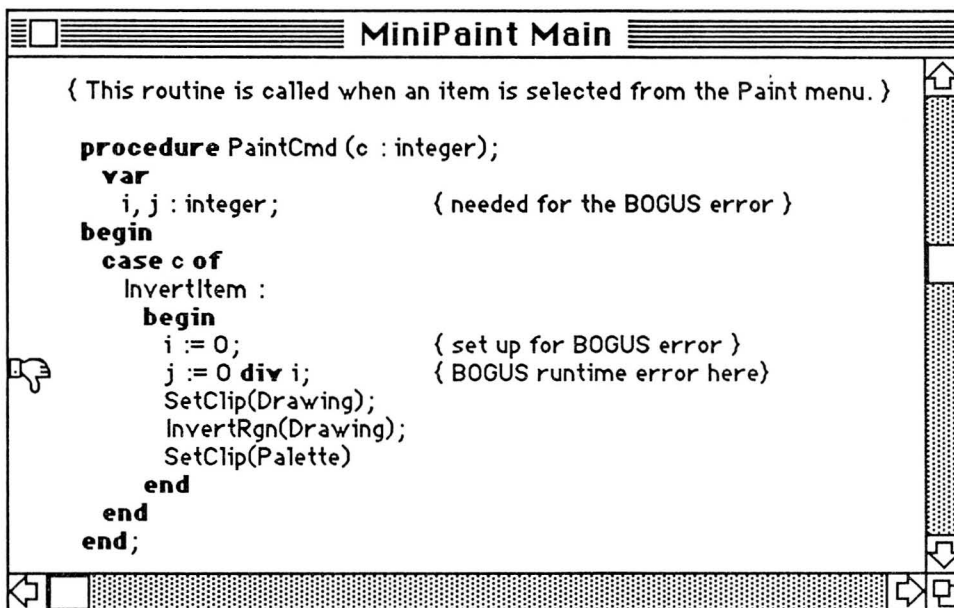
More than likely, the Drawing Window will be partially covered by the Project Window, overwriting part of MiniPaint's palette. If you restart MiniPaint from the point at which it was halted, the initialization part of MiniPaint in *Init Unit*, which draws the palette, will not be re-run. In order to insure that it is executed, choose **Reset** from the **Run** Menu before choosing **Go** or any other run command. This command resets the program so that it will start from the beginning.

An Error in MiniPaint

We've created an intentional error in *MiniPaint* in order to further demonstrate the debugging features of Lightspeed Pascal. While *MiniPaint* is running, choose the **Invert** command from the **Paint** Menu of *MiniPaint*. The program will halt because of a run time error, and an error message will appear describing the problem.



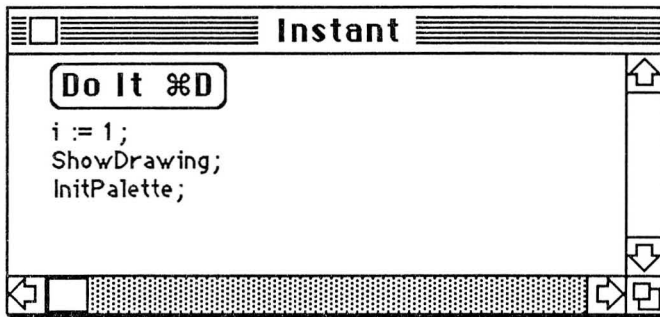
Click on the error message to make it go away. A down-turned thumb appears next to *MiniPaint Main* in the Project Window. Open *MiniPaint Main* by double-clicking on its name. An editing window for *MiniPaint Main* will appear, and after the file is "checked", a down-turned thumb appears to the left of the statement where the error occurred.



Fixing the Error - Temporarily and then Permanently

With the *Bullseye* project, you used **Instant** to make a run-time modification to a working program. Here you'll use **Instant** to fix a non-working program. By reading the source code displayed in the editing window, you can see that the value of *i* is 0. An attempt was made to divide by *i*, resulting in the run-time error.

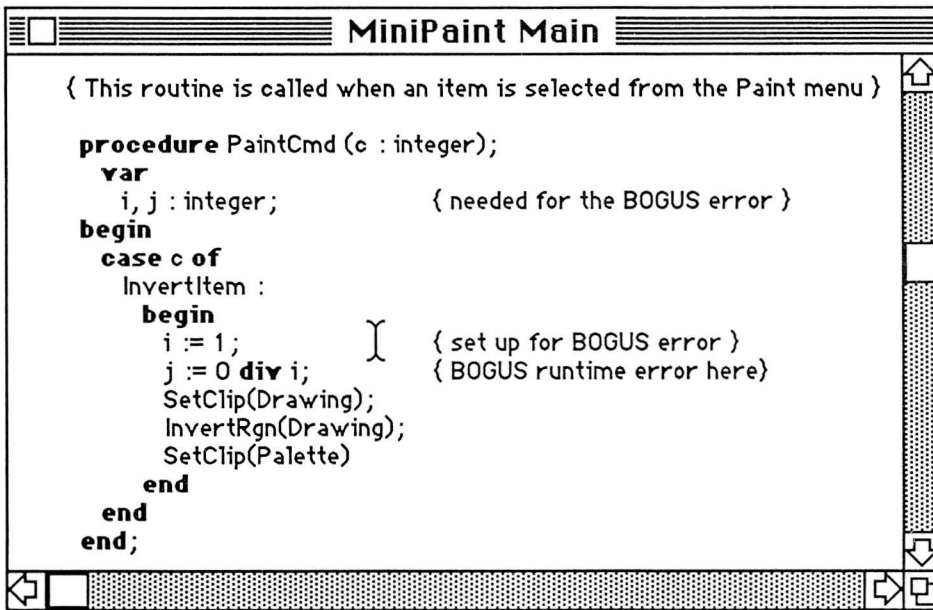
With almost any other development system, you would have to edit and recompile *MiniPaint Main*, link the whole project, and restart the program. However, in Lightspeed Pascal, you can immediately fix this error by using the Instant Window. Choose **Instant** from the **Windows** Menu, and move the Instant Window to an out-of-the-way place on the screen. Set the value of *i* to 1, reset the Drawing Window with a call to *ShowDrawing*, and reset the palette with a call to *InitPalette*. Your Instant Window should look like this:



Click the **Do It** button, and the Drawing Window will come forward. The MiniPaint palette will be re-initialized and the entire drawing area will be inverted.

You were able to fix a run-time error and continue without recompiling your program!

However, the program still hasn't been fixed—you have only patched around the problem. Choose **Invert** a second time, and the program will halt with the same error as before. You can make a temporary fix again by clicking on the **Do It** box in the Instant Window, but to really fix the program you must edit the file. This time, when the down-turned thumb appears in the editing window, change the statement above the thumb to set *i* to 1 instead of 0.



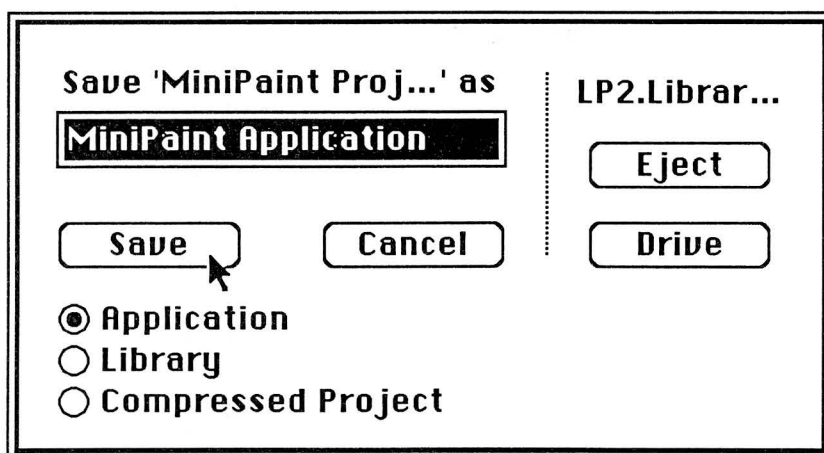
Choose **Go** from the **Run** Menu. Lightspeed Pascal recognizes that it has to recompile *MiniPaint Main* since it has been edited. Notice that only *MiniPaint Main* is recompiled, since Lightspeed Pascal knows that only this file has changed.

Before running the program, a box asks you whether to save the editing changes to *MiniPaint Main*. Lightspeed Pascal is trying to protect you from a situation where you lose your changes because of an improperly executing program. In this instance, click the **No** button, so that the demo program retains its error for the next time you go through the demo. The change will remain in the program until you quit Lightspeed Pascal.

This time, the **Invert** command should work properly.

Saving a Project as an Application

Since the project runs correctly, you can make it into a double-clickable application that will run independently from the Lightspeed Pascal development environment. Choose **Build & Save As...** from the **Project** Menu. A box will request that you give the application a name. Enter a name, and specify that you want to build an application, rather than a library or compressed project.



The project will be recompiled and built as an application. After leaving Lightspeed Pascal, you can start up MiniPaint by double-clicking on its application icon.

Leaving Lightspeed Pascal

To exit from Lightspeed Pascal, choose **Quit** from the **File** Menu. A box will ask if you want to save your changes to the project. Click the **No** button.

Where to Go from Here

These two sample programs have covered many of the features of Lightspeed Pascal. The next step depends on what you already know, and what you want to accomplish.

You may wish to start creating your own Pascal programs. You'll need to know how editing in Lightspeed Pascal works, which is described in the next chapter.

There is another demo project included on the LP2.Libraries disk, *Editor Project*. This program creates a text editor similar to other popular Macintosh editors. It is a larger program that utilizes many standard features of Lightspeed Pascal and the Macintosh. You may want to build this project, save it as an application, and actually use it as a general-purpose text editor.

If you have existing Pascal source files, you may wish to add them into projects and run them. Look over Appendix B, *Porting to Lightspeed Pascal*, for information on porting existing programs to Lightspeed Pascal.

Chapter 4

Editing

In many ways, editing in Lightspeed Pascal is like editing with other popular Macintosh editors such as MacWrite and Macintosh Pascal's editor. For example, common tasks such as selecting, copying, saving, etc. all work similarly. However, there are also significant differences, because the Lightspeed Pascal editor is an integral part of the development environment. For example, Lightspeed Pascal keeps track of which files you edit, and determines what to recompile the next time you run.

The differences fall into three major groups:

1. File and Project Management.

- Up to eight editing windows can exist simultaneously.
- Files can be opened directly from the Project Window.
- The go-away box does not close a file, but simply hides its window.
- **Add Window** adds the contents of an editing window to the project.
- Files can be saved simply as text files, or as the entire document, which includes pretty printing and Stop Signs (see Chapter 7).
- **Save As...** removes the old file from the project, and adds the new version instead. **Save a Copy As...** simply saves another copy of the file, without affecting the project.

2. Pascal-Specific Features.

- Pascal programs are pretty printed as they are entered.
- Syntax is incrementally checked as the program is entered.

3. Miscellaneous.

- Double-clicking selects an entire word.
- Triple-clicking selects an entire line.
- The search and replace commands **Find What...**, **Find...**, **Replace** and **Everywhere** are implemented as in Macintosh Pascal.
- **Show Selection** (or the Enter key) scrolls to the currently selected text.
- **Show Error** scrolls back to the point where Lightspeed Pascal reported an error.
- **Undo** is not supported by Lightspeed Pascal, but is provided for compatibility with desk accessories that support it.

These features are explained in detail below.

Multiple Windows

In Lightspeed Pascal, editing is done in a *Pascal Editing Window*. You can open up to eight files at a time, putting each file into an individual editing window. These windows are not just for editing. During compilation or execution, Lightspeed Pascal uses these windows to show diagnostic information. While running, you can use them for source level debugging.

Other types of windows, such as the Instant Window or the Observe Window (see Chapter 7), can be edited, and their contents copied into edit windows, or vice versa. The multiple uses of windows will become more natural to you as you gain experience with Lightspeed Pascal.

Opening an Editing Window

You can open an editing window in one of several ways:

- 1) To create a new file, choose **New** from the **File** Menu. A blank, untitled editing window will appear. You can edit and save the file, but you cannot compile it until you add it to a project.
- 2) To edit an existing file, choose **Open...** from the **File** Menu. A dialog box will allow you to select which file to open. Again, the file must be added to a project before you can compile it.
- 3) Once a file has been added to a project, you can open it simply by activating the Project Window (see below) and double-clicking on the filename.

When opened, the editing window becomes the active window by default.

A limit of eight editing windows can be open at one time. If you reach this limit, simply close a window before opening the next file. (To close a window, see below).

Anatomy of an Editing Window

While the editing window itself is quite similar to the ones used by MacWrite and other text editors, a brief pictorial review is not out of place. Here's what to look for:

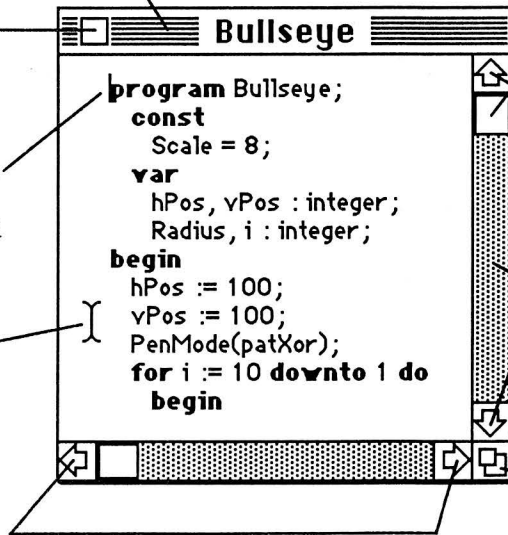
The title bar. Drag the bar to move the window around on the desktop. The title bar is always highlighted like this when the window is active.

The go away box. Click here to hide the window.

The insertion point. The characters you type will be inserted here.

The pointer. Click the pointer to set the insertion point or drag it to select text.

Scroll arrows scroll from side to side.



The scroll box. Shows where the window is relative to the beginning and end of your program. Drag the box to scroll quickly towards either end.

Scroll arrows. Click the arrow pointing in the direction you want to scroll the window a line at a time. Press the arrow to scroll continuously.

Click the gray area to either side of the scroll box to scroll a windowful at a time.

The size box. Drag the box to make the window larger or smaller.

Activating an Editing Window

Only one window at a time can be active (in front, and available for work). A window can be activated by one of three methods:

- 1) Clicking on the visible part of a partially hidden window .
- 2) Choosing the name of the window from the **Windows Menu**.
- 3) Double-clicking in the Project Window on the name of a file which is already open. (If the Project Window is not active, you will actually have to triple-click, once to activate the Project Window, and twice to activate the file.)

When you activate a window, all the other windows are *deactivated*, or *placed in back*. A window can be reactivated by any of the methods described above.

A window can also be *hidden* by clicking on the *go away* box in the upper left corner of the window's title bar. Note that the file in a hidden window is *still open*. Hiding it does not close the file, but merely hides the window. To close a file and remove its file window, you must activate it from the **Windows Menu** and explicitly close it.

You can also move a window around without activating it. Hold down the command key (**⌘**) while dragging the window's title bar.

Closing a File

In order to close a file, its window must be the active window. Choose **Close** from the **File** Menu. If you have made any changes to the file, Lightspeed Pascal will ask you whether you want to save the changes. Click **Yes** or press Return to save the changes, click **No** to close the file without saving the changes, or click **Cancel** if you change your mind about closing the window.

Saving Your Changes Without Closing

Lightspeed Pascal offers three different commands on the **File** Menu for saving. While all three save your changes, they have different effects on the project .

Save

This command simply saves the current version of your file.

Save As...

This command saves the current version of your file into a new file. The name of the active window changes to the new file name, and all subsequent edits will affect the new file.

If the file you were editing is in the current project, Lightspeed Pascal preserves the association with that project. The file's entry in the project is changed to the new file name, assuming that another file by that name is not already in the project. If a file by that name already exists in the current project, you will be asked to choose a different name for the file. You are not allowed to overwrite an existing project or library file. You have to delete it first.

You are also given a choice between saving the file as Text Only or as an Entire Document. The form in which you save your files depends on how you want to use them. Files saved as "Text" take up less space, and can be edited with other editors, but take longer to load into Lightspeed Pascal. Files saved as "Entire Document" preserve Stop Signs (described in Chapter 7) introduced into the file, and load faster, but take up more space on your disk. **Save As...** and **Save a Copy As...** allow you to change the form in which the document is saved as.

Save a Copy As...

Unlike **Save As...**, this command does not affect the project or the editing window; it simply "snapshots" the contents of the file in the window to another file. This is a good way to make backup copies without mistakenly editing the backup. The original file remains in the active editing window and the project is unchanged.

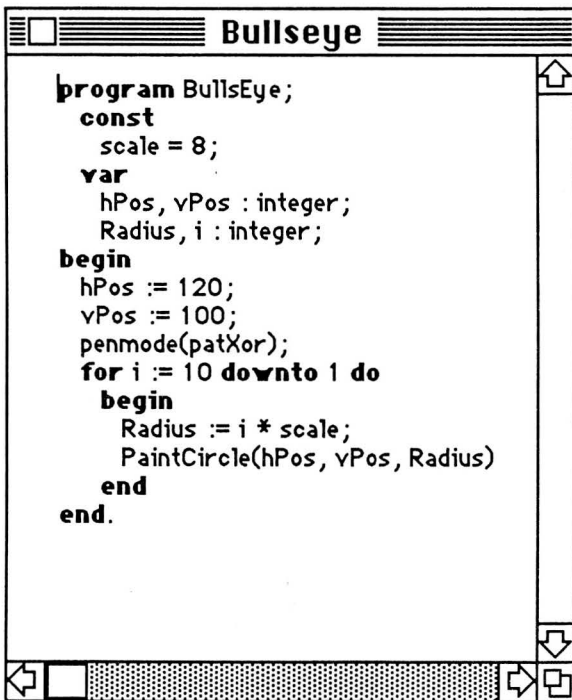
Pascal-Specific Features

The Lightspeed Pascal editor is not just another text editor, but an integrated part of the whole Pascal development environment. The two Pascal-specific features are pretty printing, and incremental syntax checking.

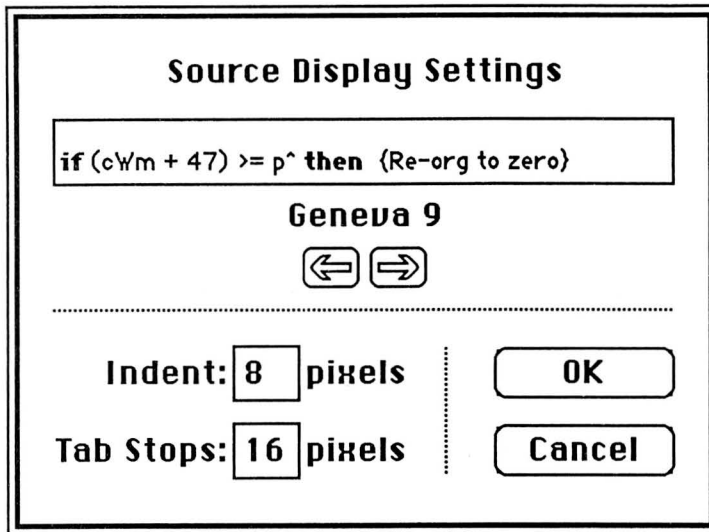
Pretty Printing

As you Enter Pascal source code in an editing window, it is automatically formatted or *pretty printed* when you type a semicolon or press Enter or Return.

This means that Pascal reserved words such as **program**, **begin**, and **end** are displayed in lowercase boldface type and that certain lines are indented and reformatted in order to make the code easier to read. The example below illustrates how indentation is used to show the current nesting of structured statements:



The amount of space each line is indented when pretty printed, and how far the Tab key spaces over, can be controlled by choosing the **Source Options...** command from the **Project** Menu.

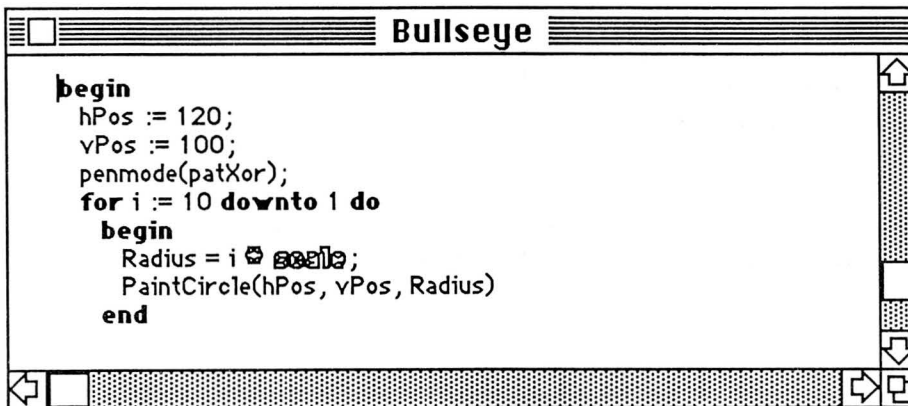


If you want to disable the indenting feature of Lightspeed Pascal, set the value of **Indent** to 0 pixels. It is not possible to disable other aspects of pretty printing.

The **Source Options...** command in the **Project** Menu is also used to choose the font in which program text is displayed. Click the right and left arrows to cycle through the available fonts and point sizes. A sample line is displayed in the chosen font.

Syntax Errors

Lightspeed Pascal checks the syntax of your Pascal program as you enter it. The syntax is checked when you type a semicolon, press Enter, press Return or click on another line in your program. If a syntax error is found, the rest of the line after the error is redisplayed in outline. In the example below, the ":" is missing from the Pascal assignment operator, `:=`.



Once you have corrected the error, you can just go on editing. The outlining will disappear as soon as you next press Return or Enter, or click on another line in the file.

A Special Purpose Editor

One implication of the editor's pretty printing and syntax checking is that you cannot use it to create anything other than Pascal program text. For example, if you want to write an RMaker script to create a resource file (see Appendix H), you must use another editor. (For this reason, one of the sample projects included on the distribution disks is a general purpose text editor. You may well want to build this editor and save it as an application.)

If you want to use another editor to edit source files created with the Lightspeed Pascal editor, you should be sure to save them as Text Only, not Entire Document.

Likewise, if you want to use the Lightspeed Pascal editor to edit files created with MacWrite or other editors, you should be sure to save them as Text Only.

Miscellaneous Goodies

The text editor includes a number of miscellaneous features that we thought would prove helpful. These include some extended selection techniques, and a slightly non-standard (at least to MacWrite users) implementation of the search and replace function.

Selecting Words and Lines

You can select a word by double-clicking on it. You can select an entire line by triple-clicking on it. If you drag after double- or triple-clicking, the selection will be extended by word or by line.

Moving to the Selection Point

Lightspeed Pascal provides the ability to examine other areas of the text, then jump back to where you were. Pressing the Enter key or choosing the **Show Selection** command from the **Edit** Menu while you are anywhere in the text will reposition the text to show the current insertion point or the start of the selected text.

Moving Back to an Error

If Lightspeed Pascal discovers an error in your Pascal program, and the editing window where the error is located is open, then that window will become the active window, and a down-turned thumb will point to the line with the error. (This feature is described in more detail in Chapter 6.)

If you scroll the window away from that point, you can quickly Return to that point by using the **Show Error** command from the **Edit** Menu.

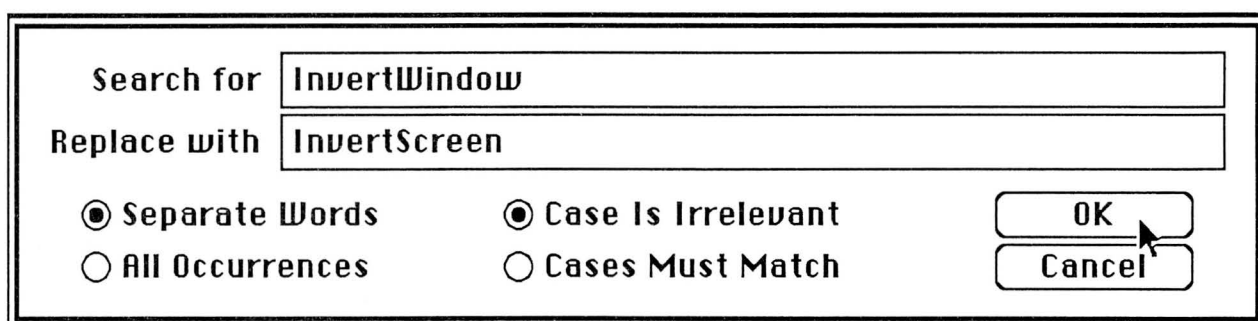
Search and Replace

Search and replace functions are available in the **Edit** Menu rather than on their own menu as in MacWrite. These functions are implemented as in Macintosh Pascal, instead of as in MacWrite.

You can use all search and replace functions in any editing window as well as the Instant and Observe windows. Each choice applies to the active window.

To enter a search string, choose **Find What...** from the **Edit** Menu. This command only sets up the conditions for the search (and optional replacement). It does not actually initiate a search.

The box allows you to enter a search string only, or both a search string and a replacement string, as shown below.



Search for

Replace with

☒ **Separate Words** ☒ **Case Is Irrelevant**

☐ **All Occurrences** ☐ **Cases Must Match**

There are also several options. To have the search locate text that matches the text you type in only if it is surrounded by "word separators" (spaces or punctuation), click the **Separate Words** button.

To have the search locate all the text that matches the text you type without regard for word separators, click the **All Occurrences** button.

To have the search disregard uppercase and lowercase as a criteria for matching text, click the **Case is Irrelevant** button.

To have the search consider case as a basis for matching text, click the **Cases Must Match** button.

- Note:**
- 1) The **Separate Words** searching mode has been optimized for speed, and will in general be many times faster than when **All Occurrences** is selected.
 - 2) Searching starts at the insertion point, only searches forward, and stops at the end of the file. The insertion point is not necessarily displayed in the visible text.

When you have everything set up the way you want it, click **OK** or press Return, which will remember your search and replace text, and any options you have selected, and will put the dialog box away.

If you change your mind about what you've done, click **Cancel**. The contents of the "What to Find" window will revert to what they were before, and the dialog box will be put away.

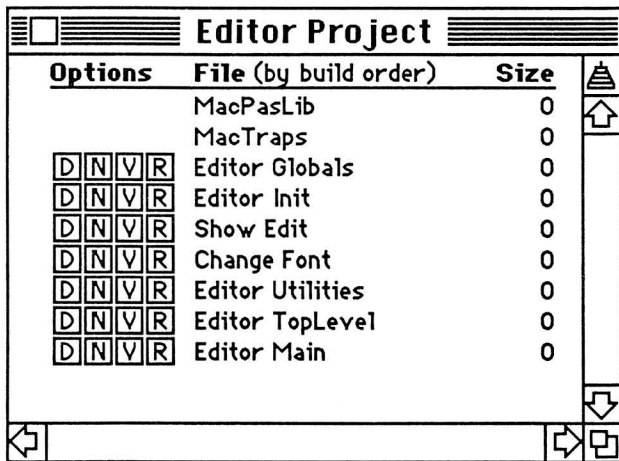
To actually initiate a search, use the **Find** command. To make a replacement, use the **Replace** command. To make a global replacement, use the **Everywhere** command.

Chapter 5

Projects

The Pascal statements that comprise a complete program can exist in many files. It is the Project that creates a framework within which the files exist as a program. In essence, running a program involves compiling, linking and executing the files contained in the Project.

The Project Window gives a visual display of the information concerning each of the files in your program. By default, this information includes the filename, which compile options are enabled or disabled for that file, and the size of the object code associated with each compiled source file. You can also list additional information (which we'll get to in a moment). For example, here's the Project Window for the Text Editor demo on the distribution disks:



The screenshot shows a window titled "Editor Project". It contains a table with three columns: "Options", "File (by build order)", and "Size". The "Options" column contains a grid of checkboxes labeled D, N, V, and R. The "File" column lists the files: MacPasLib, MacTraps, Editor Globals, Editor Init, Show Edit, Change Font, Editor Utilities, Editor TopLevel, and Editor Main. The "Size" column shows the size of each file in bytes, all of which are 0. There are also some icons in the right margin of the table.

Options	File (by build order)	Size
	MacPasLib	0
	MacTraps	0
D N V R	Editor Globals	0
D N V R	Editor Init	0
D N V R	Show Edit	0
D N V R	Change Font	0
D N V R	Editor Utilities	0
D N V R	Editor TopLevel	0
D N V R	Editor Main	0

When a project is first created, it contains the two default library files *MacPasLib* and *MacTraps*. *MacPasLib* contains code for all the predefined routines that are part of Lightspeed Pascal. *MacTraps* has "glue" to parts of the Toolbox. Files can be created in editing windows and added to the project using the **Add Window** command on the **Project** Menu, or they can be added directly from disk files with the **Add File...** command on the **Project** Menu. The project shown above already has all of the source files added to it.

A working project will always contain at least one source file. A large program like a text editor is likely to be made up of many files. One of these files will contain the main program—the collection of statements from which all execution begins and terminates. In addition, there are likely to be a number of *units* and/or *libraries*.

A *unit* is the smallest collection of Pascal statements (other than the main program) that is compilable. (Other programming languages refer to units as *modules* or *packages*.) A unit has two parts: an *interface* and an *implementation*. The interface declares the data types, variables,

procedures or functions that are visible to users of the unit. The implementation contains the Pascal code that implements the routines declared in the interface. (The implementation may also declare types, variables, procedures and functions, but these are private and not visible to users of the unit.) See Chapter 8 of the *User's Guide* and Section 8 of the *Language Reference* for more information on units.

Libraries are self-contained, pre-compiled object files that have previously been created with Lightspeed Pascal or with another development system. Two libraries that will be included in almost every project are *MacPasLib*, which contains the code for all of the predefined routines that are a part of Lightspeed Pascal, and *MacTraps*, which contains the code for Macintosh Toolbox calls that are not already resident in the Macintosh ROM.

The interfaces to all of the routines in these libraries are predefined. That is, you do not have to declare the interfaces to the routines; it has already been done for you. For most other libraries, you must add both the library itself, and a short source file that describes the interfaces to the routines in the library. New libraries that you create may not require this extra interface file. (See Chapter 9 for details.)

Chapter 8 provides information on the standard libraries available with Lightspeed Pascal, and Chapter 9 describes how to save your own code as a library that can be used in other projects.

What's in the Window

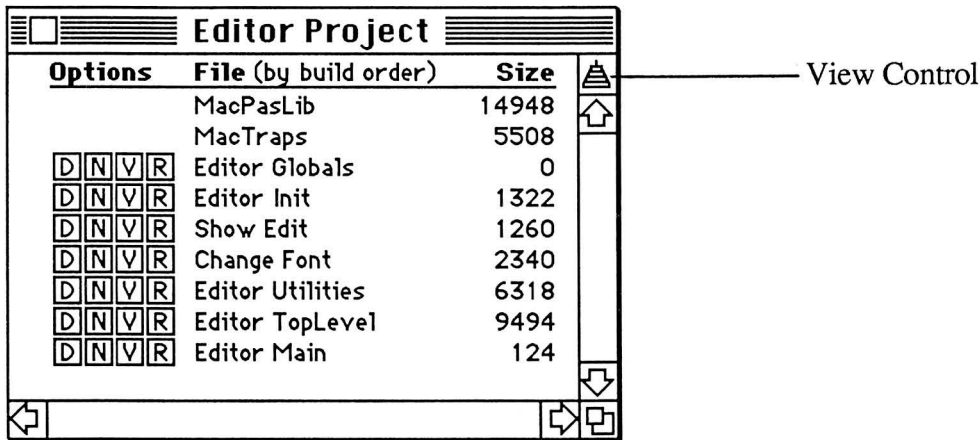
When you create a new project, Lightspeed Pascal displays three columns in the Project Window: **Options**, **File**, and **Size**, in that order. (We're going to talk about **File** and **Size** now. **Options** will be discussed at the end of this chapter.)

The **File** column displays the names of the files contained in the project. Double-clicking on a filename opens an editing window for that file. If the file was already open, its window is brought to the front. Clicking on a filename selects that file. The name can be dragged to a different location in the Project Window. We'll see the significance of that in a moment.

The **Size** column shows the size of the object code, expressed in bytes. For a newly added file, the size is zero: the file hasn't been compiled yet. A unit that contains only declarations will also have a size of zero, since there is no object code associated with it.

Once a source file has been compiled successfully, the project contains the resulting object code. There is no separate object file. The object code becomes part of the project, and shows up only as a number of bytes in the **Size** column. (Incidentally, the size of the object code shown in the Project Window is not necessarily the size of the program when it is saved as an application. When a project is saved as an application, the size may be reduced, since "smart linking" will discard unused units and/or library routines.)

Let's look again at the Project Window for the Text Editor demo, this time after all of the files have been compiled, and the object code added to the project.



Two Views of the Project

There are two ways in which files can be ordered in the Project Window: **by build order** (the default), and **by segment**. You can select between these two display options by clicking on the view control in the upper right corner of the Project Window.

When viewed **by build order** (as in the example just above), files are displayed in the order that they will be compiled. In order to build successfully, the program has to be compiled in a specific order that takes into account the dependencies between the various units it contains. For example, when your program references a procedure in another unit, Lightspeed Pascal has to have already compiled the referenced unit. (Among other things, this implies that the main program will always appear last in the Project Window.)

As you add new files to the project, they are added by default after any other files already in the project. However, you can re-arrange the order of the files by selecting them, and then dragging them around within the Project Window.

There is a second view of the project: **by segment**. You can toggle between these two views by clicking on the view control in the upper right corner of the Project Window. When viewed **by segment**, the Project Window for the Text Editor demo looks like this:

Editor Project		
Options	File (by segment)	Size
	MacPasLib	14738
	MacTraps	5764
D N V R	Editor Globals	0
D N V R	Editor Init	1322
D N V R	Show Edit	1260
D N V R	Editor Main	124
	<i>Segment 1</i>	23212
D N V R	Change Font	2340
	<i>Segment 2</i>	2344
D N V R	Editor Utilities	6318
D N V R	Editor TopLevel	9494
	<i>Segment 3</i>	15816

The Macintosh restricts the size of a code segment to 32K bytes. Lightspeed Pascal allows you to get around this restriction by grouping the files in your project into multiple segments. When you view the project **by segment**, the files in the project are displayed in segment order.

Dragging a file name in the Project Window to a position just below the dotted line at the bottom of the project creates a new segment containing only that file. Dragging a filename onto another file in the Project Window moves the first file to the segment containing the second file.

Each segment is separated in the Project Window by a line. Any segments that become empty are automatically deleted. At the end of each segment is the segment number and the maximum number of total bytes of code in that segment. Each file within a segment is displayed in alphabetical order.

The order in which files are listed in each of the two project views is completely independent. If you change the build order, the segmentation will not change. Nor will changing segmentation affect the build order.

Compile Options

The **Options** column in the Project Window displays the state of the debugging options for each file in the project. Enabling or disabling these options affects the code being generated by the compiler.

A full discussion of the effect of these options can be found in Chapter 13. Briefly, the options are as follows:

- D Debug. Enabling this option generates additional information between each Pascal statement. This code supports stepping, stopping, and observing.
- N Names. Enabling this option inserts the name of all routines (truncated to eight characters) into the code. This is useful for debugging with Lightsbug and Macsbug.
- V Overflow Checking. Enabling this option generates code that checks for integer arithmetic overflows.
- R Range Checking. Enabling this option generates code that does range checking for array indexing, assignments, and parameter passing. It also generates code that checks for nil pointer dereferencing.

A box around the one letter designation for the compile option indicates that that option is currently turned on. By default, all the options are on. To change the status of a particular compile option, move the mouse over the letter for the desired option and click. The box will appear and disappear with each click of the mouse.

If you hold down the option or command key when changing an option, the new state of that option will carry over to all files in the project.

Here's an example of a display with some of the options turned off:

Editor Project			
Options	File (by build order)	Size	
	MacPasLib	14948	
	MacTraps	5508	
D N <input checked="" type="checkbox"/> V R	Editor Globals	0	
D N <input type="checkbox"/> V R	Editor Init	1322	
D N <input type="checkbox"/> V R	Show Edit	1260	
D N <input type="checkbox"/> V R	Change Font	2340	
D N <input type="checkbox"/> V R	Editor Utilities	6318	
D N <input type="checkbox"/> V R	Editor TopLevel	9494	
D N <input type="checkbox"/> V R	Editor Main	124	

More Things to Look At

You can customize your Project Window with the **View Options...** command in the **Project** Menu controls. When you choose **View Options...**, the following dialog box will be displayed:

Project View Settings

Options	File (by segment)	Size
<input checked="" type="checkbox"/> D <input checked="" type="checkbox"/> N <input checked="" type="checkbox"/> V <input checked="" type="checkbox"/> R	MemHacks	32767
<input checked="" type="checkbox"/> D <input checked="" type="checkbox"/> N <input checked="" type="checkbox"/> V <input checked="" type="checkbox"/> R	Extremely w-i-d-e...	10

☒ filename
☒ options
☒ code size

☐ unit name
☐ volume name
☐ date file saved

File Information

Geneva 9

⬅
➡

OK

Cancel

Options - Boxed indicates enabled

D - Debug. Allows stepping, stopping, stack checking, Observing.

N - Names. Insert Macsbug names into the code.

V - Integer arithmetic overflow checking.

R - Range checking.

In addition to **File**, **Size**, and **Options**, you can display the following information for each file in the Project: **Unit Name**, **Volume**, and **Date Saved**. You can also rearrange the order in which the information is displayed.

Unit Name gives the name of the unit (i.e., the identifier specified in the *unit* declaration). This is not necessarily the same as the filename, although to avoid confusion, you may want to give your files the same names as the units they contain. The unit name does not appear until the unit has been compiled.

Volume shows you the disk or folder in which the project expects to find the file on. This can be useful information if Lightspeed Pascal can't find a file it needs to access.

Date Saved displays the last date and time you saved the file within the Lightspeed Pascal development environment.

The large box at the top of the **View Options** dialog box shows how an example project would be displayed given the current settings. Clicking on the boxes in the File Information section of the window changes what information is displayed about files in the project. The order the boxes were clicked on will be the order in which the categories are displayed. In the box shown below, we've selected *filename*, *code size*, *date file saved*, and *options*, in that order:

Project View Settings

File (by segment)	Size	Date Saved	Options
MemHacks	32767	12/27/85 10:43	D N V R
Extremely w-i-d-e...	10	01/02/86 23:07	D N V R

☒ **filename**
☒ **options**
☒ **code size**

☐ **unit name**
☐ **volume name**
☒ **date file saved**

Geneva 12

⬅
➡

OK

Cancel

File Information

Options - Boxed indicates enabled

D - Debug. Allows stepping, stopping, stack checking, Observing.

N - Names. Insert Macsbug names into the code.

U - Integer arithmetic overflow checking.

R - Range checking.

Note that you can suppress or rearrange the display of the three standard columns of information as well as add any of the additional columns of information that are available.

The **View Options...** dialog also allows you to change the font used for the Project Window by clicking on either of the two arrows in the middle section of the dialog box. These will cycle you through the fonts and sizes available on your system. A sample of the current font is shown in the box above the arrows.

Chapter 6

Running a Program

This chapter describes how to run a program in the Lightspeed Pascal programming environment. It also describes what goes on behind the scenes when you run a program.

Before Running a Program

If your program uses the Text and/or Drawing Windows, and does not open them automatically (using ShowText and/or ShowDrawing calls), you will have to open them explicitly before running the program. Use the **Text** and **Drawing** commands in the **Windows** Menu to open those windows.

Running a Program

Normally, to run a program, just choose **Go** from the **Run** Menu. The other run commands, **Go-Go**, **Step**, and **Trace**, become important when trying to debug your program: they're described in detail in Chapter 7.

Here's What You See

If you have not yet run your program, or if any of the files in the project have been changed since you last ran your program, Lightspeed Pascal will first compile and link your program. You will see a "speedometer" box as each file that needs compilation is compiled.

If you have not saved your files, you may be asked if you want to save the files before running the program. This reminder can be suppressed in two ways: one which will automatically save your files prior to running, and one which will assume you *don't* want to save your files before running (see *Save Options*, below).

Once the program starts running, a Bug Spray Can appears on the right side of the menu bar.

Behind the Scenes...

The following is a more detailed explanation of what is happening when you choose any of the run commands:

- 1) Lightspeed Pascal does selective recompilation. It checks the interface for each file in the program which has been modified, and tags for compilation any files affected by a change in the interface. If this is the first time the program has been run, *all* the files in the project are tagged for compilation.
- 2) The tagged files are compiled in *build order*. When you view the Project Window by build order, the top file is the first to be compiled.
- 3) *Linking* of your program occurs next. This checks that all names referenced across files are in fact defined somewhere, and that they are defined only once.

Lightspeed Pascal's linking process takes advantage of architectural features of the Macintosh. Rather than having to resolve *all* references during the link process, it only has to resolve a few selected references. As a result, the linking process is very fast, even for a large program.

- 4) Assuming the previous steps occur smoothly, your program is loaded into memory and starts running. The Bug Spray Can appears in the menu bar on the right hand side, and disappears when the program finishes.

When Something Goes Wrong

If an error occurs during the any of these steps, a box will appear summarizing what Lightspeed Pascal thinks went wrong.

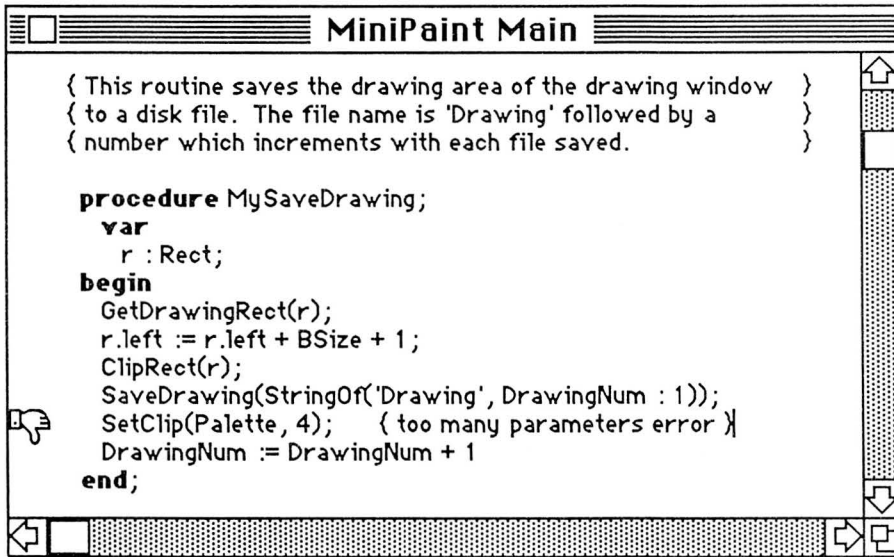


Too many parameters used in procedure or function call.

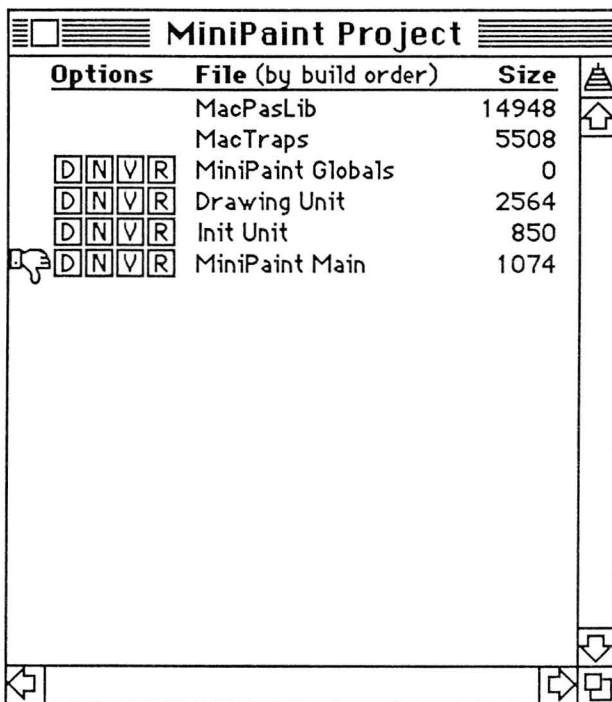
The message may be different, but the bug illustration will always be there. You can click anywhere on the box to make it go away, but you still need to fix the problem.

Pascal Statement Errors

If you program has an error in it, Lightspeed Pascal will point out the error with a "thumbs down" next to the offending line in the file's editing window.



However, if the error occurs at run time and the file's editing window is not open, the downturned thumb will appear in the Project Window, next to the name of the appropriate file.



Other Errors

Other errors can occur. For detailed explanations of Lightspeed Pascal error messages, see Appendix F, *Error Messages*.

Compiling, Building or Linking Without Running

The compile and link steps described above can be performed alone as follows.

The **Check** command compiles only the file in the active window. It is useful for quickly checking that your Pascal syntax is correct.

The **Build** command *builds* the project, i.e., performs all of the steps necessary to prepare a project for execution without actually running the program. Compilation errors found during the build are signaled with an alert box and a "thumbs down" next to the offending line. A window is opened automatically if a compile error occurs during a **Build**. Link errors are signaled with an alert box.

The **Check Link** command in the **Run** Menu performs a build, and then links your program.

Save Options

If you don't save changes to your files before running and your program crashes, you may lose all of your unsaved edits. As previously mentioned, Lightspeed Pascal checks to see if any files have been edited without being saved before trying to run your program. The Save Options in the **Run** Menu let you specify whether or not you want saving to occur, and whether or not you want to confirm the **Save**.

There are three different choices for this option:

Choosing **Auto-Save** will cause all unsaved edits to be automatically saved without any confirmation. This state is safe and unobtrusive, but you may inadvertently replace previous versions of your files with newly modified versions.

Choosing **Confirm Saves** directs Lightspeed Pascal to ask for confirmation before saving each edited file. This allows trial edits to be executed without replacing previous versions. This is the default state.

Choosing **Don't Save** will cause Lightspeed Pascal not to save changes before execution. This option implies that you are willing to risk losing your edits in exchange for the time savings of not having to write out files.

Halting a Program

You can stop a program anytime by clicking on the Bug Spray Can on far right end of the menu bar.

To continue execution, choose any of the run commands (**Go**, **Go-Go**, **Step**, **Trace**) again from the **Run** Menu.

To restart the program from the beginning, choose **Reset** from the **Run** Menu before choosing **Go**. An implicit **Reset** is done when you edit any file in the project, change any compile options, or change the project view. Note that you can only stop your program when it is running object code that has been compiled with the Debug option on. If it is running in code that was compiled with Debug off, it will stop at the first place it encounters code that was compiled with Debug on.

Run Options

Lightspeed Pascal allows you to set up certain run-time environment settings. Choose **Run Options...** from the **Project** Menu. A dialog box like this will appear:

Run-time Environment Settings

Resources

☐ Use resource file:

for resources used by the project.

Text Window

Text Window saves characters

☐ Echo to the printer

☐ Echo to the file:

↔

Monaco 9

Hello world. x = 811.79.

Memory

Stack size: kilobytes

Zone size: kilobytes

OK

Cancel

These settings affect the following areas of the run-time environment:

- Resources

To designate a resource file to be used by your program, click on the **Use resource file:** check box and choose a resource file to use. This resource file will be automatically opened when your program is run. For more information see Chapter 10, *Using Resources*.

Text Window To set the number of characters that Lightspeed Pascal preserves in the Text Window, type in the desired number in the **Text Window saves** box.

To echo all Text Window output to the printer, click on the **Echo to the printer** check box.

To echo all Text Window output to a file, click on the **Echo to the file:** check box and choose a file to use.

You can also change the font of the characters displayed in the Text Window. You can browse through the available fonts and sizes until you find the one you want by clicking on the two arrow buttons. The current font name and size are displayed along with a sample of text in that font.

Memory To set the amount of memory reserved for your program's stack, type in the desired number in the **Stack size:** box. The amount of memory your program needs for a stack is based on the maximum amount of storage needed for local variables and stack frames. The default stack size is 16K bytes. If your program does not have many local variables and is not deeply recursive, you may want to decrease your stack size to free up some extra memory. If your stack size is too small, you will get a stack overflow error.

To set the amount of memory reserved for your program's zone, type in the desired number in the **Zone size:** box. The amount of memory your program needs for a zone is based on the maximum amount of storage needed for your program's code together with the memory needed for dynamically allocated objects. The default zone size is 64K bytes. If your program is not very large and does not use much dynamic memory, you may want to decrease your zone size to free up some extra memory for tasks such as opening Lightspeed Pascal's debugging windows.

For more information on stacks and zones, see Chapter 11, *Interfacing with Assembly Language*.

When you are done setting up the run time environment, click on the **OK** button or hit Return. If you've made a mistake or changed your mind, click on the **Cancel** button.

Chapter 7

Debugging Your Pascal Program

When you're writing a program, things don't always work the way you would expect. This can be due to simple typographical errors, making invalid assumptions when designing or coding the program, or failure to think through exactly what you meant to do.

There are several distinct classes of errors you can encounter when working with Lightspeed Pascal:

- Language syntax errors, such as a misplaced semicolon.
- Compiler errors, such as an undeclared variable.
- Link errors, such as a missing or multiply defined symbol.
- Runtime errors, that are detected only when the program is actually running.
- "Errors of intention," in which the program runs, but doesn't quite do what you want.

As described in Chapter 4, the Lightspeed Pascal editor will catch language syntax errors, and will highlight the text that Lightspeed Pascal thinks is in error by printing it in an outline font.

Likewise, compiler errors are detected when you try to build or run the project. As described in Chapter 6, when Lightspeed Pascal detects this type of error, it will display a message and will open the source file in an editing window with a downturned thumb next to the offending line in the program. (See Appendix F for a complete list of error messages.)

Linker errors are detected when you try to run the project. Lightspeed Pascal displays a message indicating which undefined or multiply defined symbol is causing a problem.

In this manual, the term debugging only has to do with the last two classes: runtime errors, and errors of intention.

Runtime errors can run the gamut from obvious errors like data out of range, or the divide by zero error intentionally introduced into the *MiniPaint* demo shown in Chapter 3, to unpredictable behavior or a total program crash.

As with compiler errors, Lightspeed Pascal will display a message and a downturned thumb next to the source line on which the error occurred. However, the point at which the error was encountered may not be the actual point at which it was introduced into the program. Such errors can be difficult to track down, simply because they are not necessarily related to any obvious

failing in your program, but may depend on user input or other conditions that occur outside of your control.

And of course no compiler can catch errors of intention. The problem is that computers are very literal. They do exactly what you program them to do. In programming it is sometimes difficult to consider every contingency. The basic trick of debugging is not just to find errors. It is to find out what your program is really doing, instead of what you thought you told it to do.

What do you do when your program crashes? Or worse, what do you do when it acts unpredictably? In the past, you had few alternatives. You could print out your program and study it carefully in the hope that the error would jump out at you. Or you could run the program under control of an assembly-language debugger like Macsbug while trying to relate the 68000 code it displays to your own Pascal listings.

What you really want is a system where you can watch your program in operation—to step through your program, statement by statement if necessary, while simultaneously watching the source code, the program's output, and the value of variables and expressions that you suspect might be involved in the error.

This programmer's dream is just what Lightspeed Pascal provides. No longer does your source code lie dormant while your program runs amok—you have a chance to study your running program on the level at which you originally wrote it. Even more, you can make temporary patches to your program and continue execution without recompiling!

Note: The Lightspeed Pascal debugging tools can only be used on files that have been compiled with the Debug (D) compile option turned on. This option, which is on by default, generates extra code that supports the debugging tools. When you **Build & Save As...** an application, Lightspeed Pascal will recompile your files with the Debug option off. Be sure to compile with Debug on if you want to **Step, Trace, Go-Go**, use Stop Signs, Instant or Observe.

Debugging in Lightspeed Pascal

We said above that the basic trick of debugging is to find out what your program is really doing, instead of what you thought you told it to do.

The first step in finding this out is to run your program in a controlled way, so that you know exactly which part of your code is being executed at a given time. Lightspeed Pascal gives you several different ways of controlling program execution. We've already seen these demonstrated in Chapter 3. You can:

- Step through your program one statement at a time, using the **Step** command (**⌘ S**). The Execution Finger will appear, pointing to the next line to be executed. Your program is halted after each statement.
- Step automatically with the **Trace** command (**⌘ T**). This command steps through your program a statement at a time. It pauses only briefly between statements, continuing execution without waiting for you to issue a step command each time.

- Insert one or more Stop Signs anywhere in your code. If the program is run with the **Go** or **Trace** commands, it will halt whenever a Stop Sign is encountered. With the **Go-Go** command, the program only pauses at a Stop Sign.
- Click on the Bug Spray Can to interrupt the program at any point.

Sometimes, just slowing the pace of program execution is enough. By watching the output of the program at the same time as you watch the source lines being executed, you can obtain an understanding of what is going on. Often, though, you need to understand not just which statement is being executed, but the values that variables and expressions take on as the program runs. For this kind of debugging, Lightspeed Pascal provides the Instant and Observe Windows.

The Observe Window works as follows. You can type in (or copy from your source file) any number of valid Pascal expressions (including simply variable names) whose value you want to know about. Their values are updated:

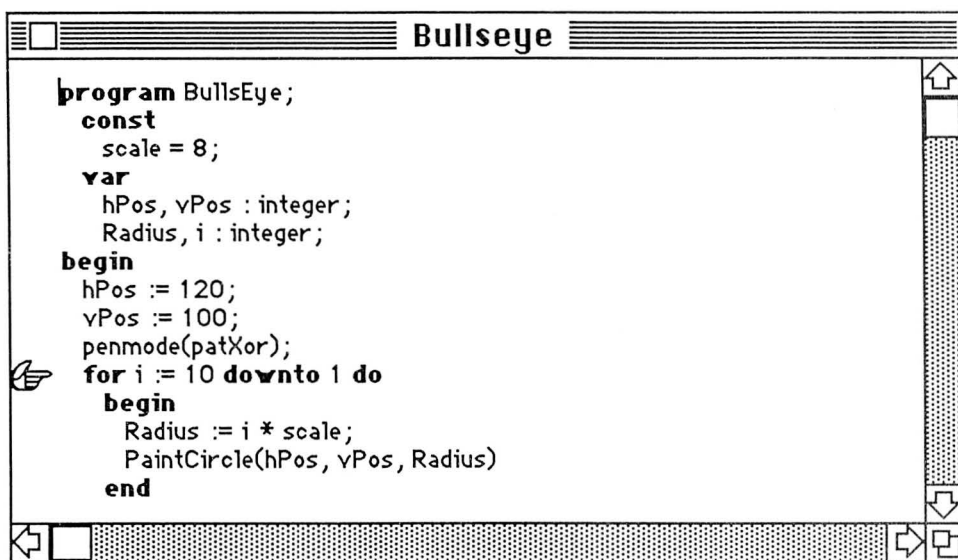
- whenever your program is halted (between steps when the program is run in **Step** mode, or at a Stop Sign when the program is run with **Trace** or **Go**).
- whenever your program pauses (between steps in **Trace** mode, or at a Stop Sign when the program is run with **Go-Go**).

If your program is already halted, you can find out the current value of a variable or expression by typing it into the Observe Window and pressing Enter.

The Instant Window takes things one step further. You can actually change the value of a variable or expression, or insert additional statements into your program, and have them executed on the spot—in the context of the program at the point at which it was halted. You can imagine the implications!

Following the Finger

The Execution Finger is the key to making use of Lightspeed Pascal's tools for controlled program execution. Whenever you step or trace, or use Stop Signs, the Finger points at each statement as it is being executed.



There are some things to know about following the Finger. First of all, a file must have been compiled with the **Debug** option on, or the Finger will not be displayed while statements in that file are being executed. If all of the files that make up the program have been compiled with **Debug** off, the finger will never appear, and none of the debugging tools described in this chapter will be available.

Second, a file that is being stepped into must be in an open editing window. If the window is not open, the Finger will point to the name of the file in the Project Window.

Third, when the Finger moves to a new window (either a file window or the Project Window), the window may be activated. If the **Auto-Show Finger** option in the **Debug** Menu is on (the default), the window is always activated; otherwise, the Finger will be displayed in the window without activating it—with the possibility that you won't see the Finger. This can actually be very useful if you want to use Observe, since the Observe Window won't be covered up when the Finger moves into a new file window.

Fourth, if the **Step Into Calls** option on the **Debug** Menu is turned off, Lightspeed Pascal will only step or trace through the current block without diving into called procedures. All called procedures will execute at full speed.

When your program is halted, you can scroll through the editing window or switch to other windows to look at other parts of your code. The **Show Finger** command in the **Debug** Menu will return you to the current location of the Execution Finger.

The **Show Finger** command activates the window in which the Execution Finger is located, and scrolls the window to the statement where the Execution Finger is pointing. If there is no editing window open for the current location of the Execution Finger, the Project Window will become the active window. In this case, the Execution Finger will point at the entry for the file in which the Execution Finger would be located.

Stepping Through a Program

To run a program one step at a time, choose **Step** from the **Run** Menu. Lightspeed Pascal will execute the first statement in the program, or if the program has already been halted, will continue with the next statement.

One statement is executed for every **Step** command. Choose **Step** again, or press **⌘ S**, to execute the next statement.

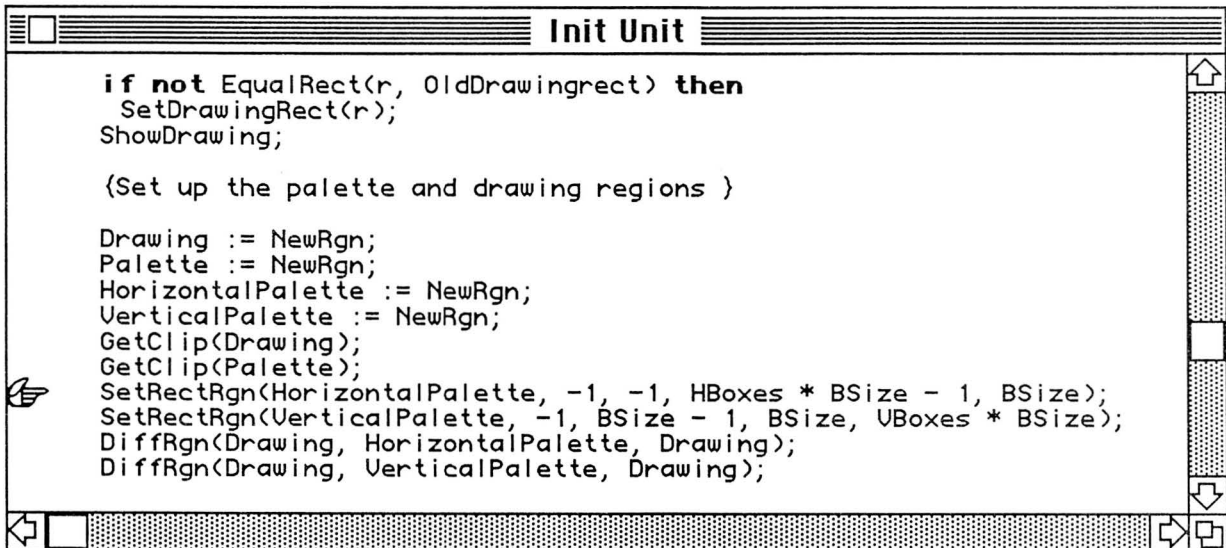
If **Step Into Calls** in the **Debug** Menu is turned off, the finger *will not* step through a procedure or function. The procedure will be executed, and the Finger will advance to the statement in the main program immediately following the call.

The Observe Window is updated with each step.

Tracing a Program

As you can imagine, stepping is appropriate only when you want to take a very close look at your program. **Trace** is the "automatic" version of **Step**. It executes your program at the rate of several statements a second: faster than you can specify with **Step**, but slow enough for you to be able to watch the program's execution. The Observe Window is updated continuously, as execution pauses briefly after each step.

Lightspeed Pascal will trace through your program until it finishes, encounters a Stop Sign, or until you halt it manually by clicking on the Bug Spray Can.



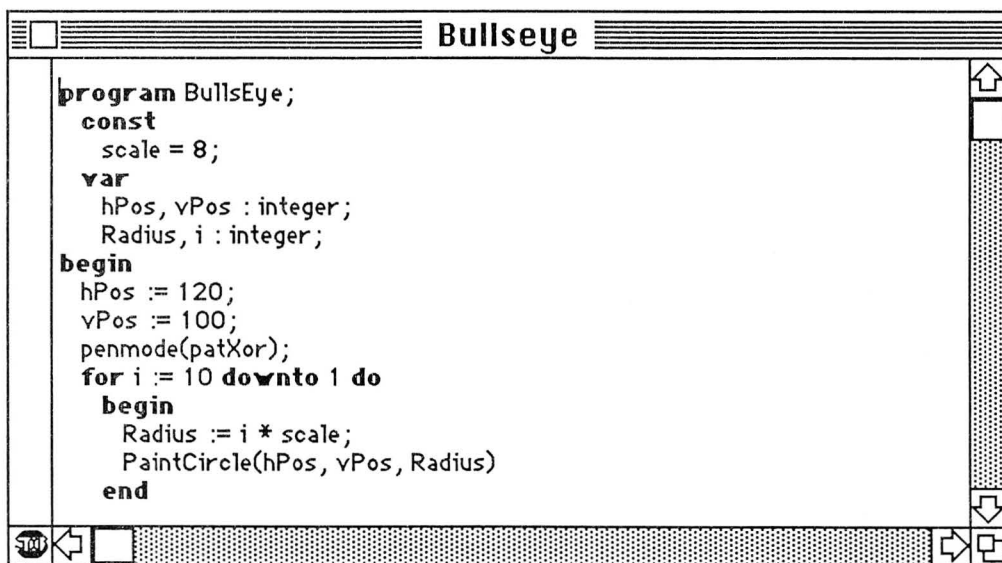
Stop Signs

Sometimes even tracing is too slow. You may want run your program at full speed up to a specific point, and then halt it. Then you either start stepping or tracing, possibly using the Observe Window.

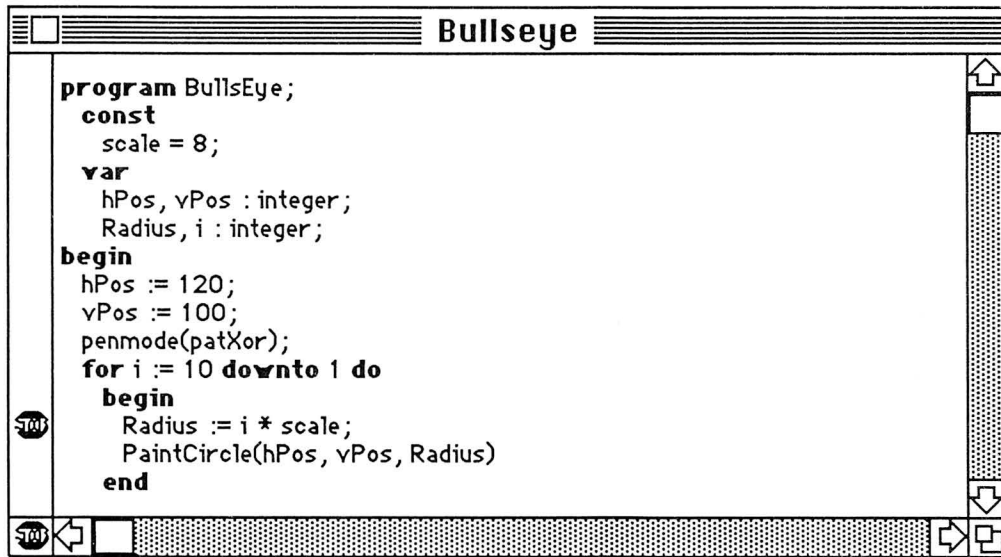
Stop Signs allow you to specify the statement(s) where you want to interrupt execution of your Pascal program. If you run your program with **Go** or **Trace**, the program will run until it reaches the Stop Sign, and then halt. The statement at the Stop Sign is not executed. If you run your program with **Go-Go**, the program will simply pause long enough at each Stop Sign to update the Observe Window, and then continue executing. (In short, using **Go-Go** with Stop Signs is much like tracing, but instead of pausing at each statement, Lightspeed Pascal pauses only at Stop Signs.)

You can only place Stop Signs on lines containing Pascal statements. Stop Signs can not be set on non-executable lines such as comments, declarations or labels. Stop Signs also cannot be placed on empty statements.

To place Stop Signs in your programs, choose the **Stops In** command from the **Debug** Menu. A Stop Sign will appear in the lower left-hand corner of the active editing window, and a vertical bar will appear along the left side.



When you move the pointer into the bar on the left side of the editing window, the pointer itself becomes a Stop Sign. Move the Stop Sign until it is directly to the left of the statement you want execution to stop before, and click. A Stop Sign will be left behind at that point, as shown for the statement beginning with *Radius* shown below.



You can put in as many Stop Signs as you want. Click on a Stop Sign to remove it.

Note: When Stops are in, a check mark appears next to the **Stops In** command in the **Debug** Menu.

If you choose **Stops In** again, any current Stop Signs will be hidden, and they will be ignored when the program executes. If you choose **Stops In** a third time, the hidden stops are redisplayed, and will be performed when the program is executed.

To remove all the Stop Signs from the active window, choose **Pull Stops** from the **Debug** Menu.

If you save a file with the **Entire Document** option, any Stop Signs placed in the file will be saved as part of the file.

Go Versus Go-Go

When you use **Go** or **Trace** to run your program, you can halt it by clicking on the Bug Spray Can on the right end of the menu bar. This halts your program whether it is stepping continuously or just running. The Execution Finger appears on the statement where you halted, and the Observe Window is updated. While this is simple to do, it is not very precise, because it is hard to halt execution accurately. (This is most useful if the program is in a loop.)

Go can be used more effectively with Stop Signs. Place Stop Signs at the exact points where you want execution to halt, and then choose **Go**. The program will run until it encounters the first Stop Sign; you can then use the Observe Window to check the values of variables at that point. Choose **Go** again, and the program will run until it reaches another Stop Sign or finishes.

Any time execution is halted, you can use **Show Finger** to scroll to the statement that you're halted at; both the Execution Finger and the Stop Sign will be at the same statement.

Go-Go is a variation of **Go** where execution pauses momentarily at each Stop Sign just long enough to update the values in the Observe Window. If there are no Stop Signs in the editing window, **Go-Go** is equivalent to **Go**.

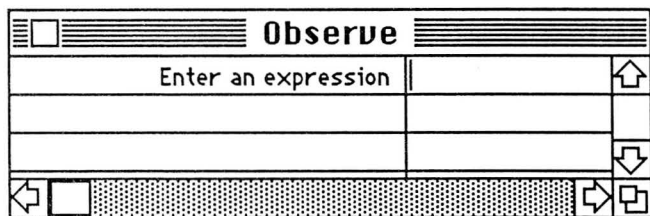
Restarting a Halted Program

Halting a program doesn't mean that you have to start it all over from the beginning. If you choose **Go**, **Go-Go**, **Step**, or **Trace** from the **Run** Menu, the program continues execution from the point at which it was halted. Only if you edit your program after halting, change the status of a compile option, or choose **Reset** from the **Run** Menu does the program start execution over from the beginning.

The Observe Window

The Observe Window allows you to track the value of a variable or expression while running your program. This window takes the place of the old debugging technique of inserting *writeln* statements into your program to print out the values of variables or expressions. (And in fact, only values that could be printed with *writeln* can be observed.)

To enter an expression in the Observe Window, bring the window forward by clicking on it or by choosing it from the **Windows** Menu.



You can place valid Pascal expressions in the "cells" to the right of the vertical bar in the middle of the window. You can even **Copy** an expression from your program, the Instant Window, or another Observe Window cell and **Paste** it into the Observe Window. The expressions will be evaluated and the results displayed in the left cells whenever you press Enter or when your program halts or pauses:

- 1) After a **Step**.
- 2) In between statements while running with **Trace**.
- 3) Whenever a Stop Sign is encountered while running with **Go** or **Go-Go**.
- 4) When you click on the Bug Spray Can while your program is running.

In order for Observe to work, there must be a context for the expression(s) you type in. That is, your program must be halted, so the expression can be evaluated for its value at that point in your program. Once the program finishes, the Observe Window will display the message "No context."

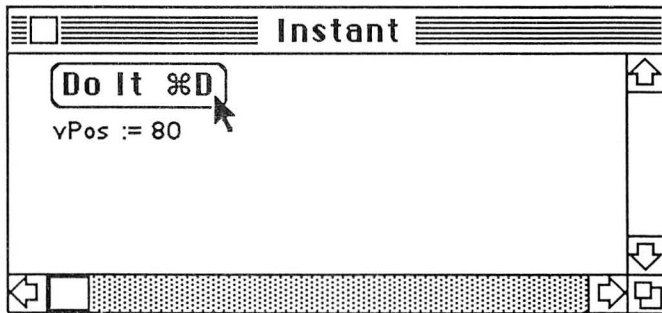
It is relatively harmless to type invalid expressions or expressions with runtime errors in the Observe Window. An abbreviated error message will be displayed in the left hand cell.

Some handy tips for using Observe:

- To observe floating point numbers with precision, use the *stringof* function described in Section 10.7.5 of the *Language Reference*.
- You can use Observe to convert hexadecimal numbers to decimal. Type '\$' followed by the hex number. The equivalent decimal value will be displayed.
- You can use the Observe Window to examine variables and expressions if execution is stopped due to a runtime error. This powerful debugging feature can help you to determine the cause of the runtime error.

The Instant Window

Any time your program is halted, you can use the Instant Window to execute any Lightspeed Pascal statement or statements. Bring the Instant Window forward by clicking on it or choosing it from the **Windows** Menu. Enter and edit any Pascal statements there just as you would in an editing window. Then click the **Do It** button or press (**⌘ D**) to execute them.



If you use (**⌘ D**), you can run the code in the Instant Window without making it the active window.

Any Pascal statement (except a *goto*) is valid in the Instant Window if it is legal at the point in the program at which execution is halted. The statements in the Instant Window are executed as if they were inserted into your program at exactly the point where execution was paused. Instant can even be used after a run-time error occurs. You can possibly fix the cause of the error and continue executing your program.

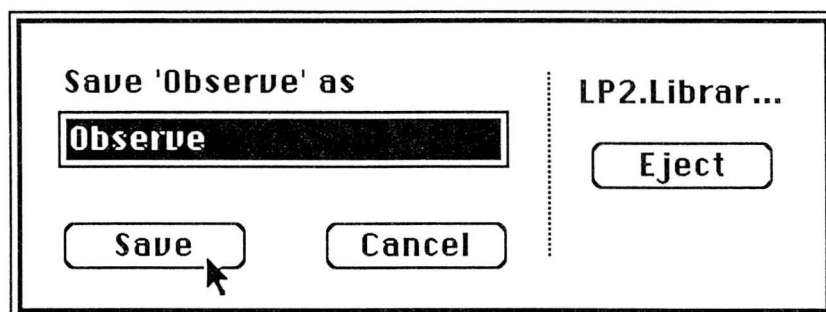
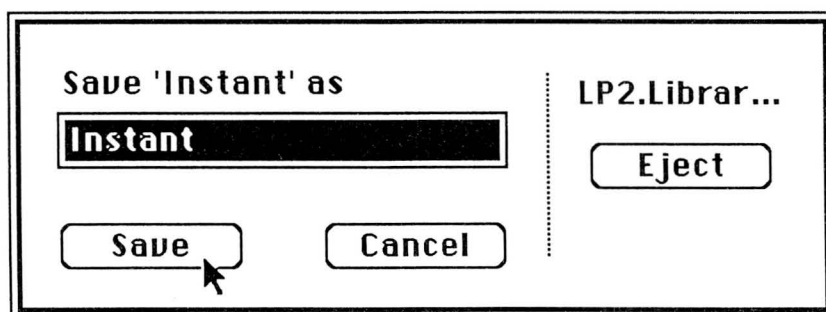
The Instant Window is also an ideal way to prototype new code. The new code can be inserted in the Instant Window and executed, and if incorrect or incomplete, modified again. This can be repeated a number of times until the code is correct. Then the new code can be copied from the Instant Window and pasted into the program. Making the changes in the Instant Window is a lot faster than continually changing the program, recompiling, and re-executing the program to try out new code. As fast as the Lightspeed Pascal compiler and linker is, Instant is even faster.

Note: If you call a procedure containing Stop Signs from **Instant**, the Stop Signs are ignored.

Don't be overly concerned about errors in Pascal syntax or expressions which you make in the Instant Window. The effects of any errors you make are limited to the current execution of your program. They are not saved as part of either your source file or the object code in your project.

Saving/Reloading Instant & Observe

The contents of **Instant** and **Observe** Windows can be saved as separate files. Choose **Save As...** from the **File** Menu when either of these windows is the active window. A special dialog box for saving these windows will appear.



Instant and Observe files appear on the disk with their own special icon.

They are opened from within Lightspeed Pascal just like any other disk file. Use the **Open...** command from the **File** Menu to reload either window. When you select a previously saved Instant or Observe file, a dialog box will appear asking if you want to replace the current contents of the Instant or Observe Window.

Chapter 8

Predefines, Units and Libraries

So far, we've looked primarily at Lightspeed Pascal's user interface. By now, you should have some feel for how to use the tools the Lightspeed environment provides. At this point, we need to explore briefly the question of what to put into the programs you write--the special features of the Lightspeed Pascal language that you can use to write powerful programs for the Macintosh.

These features include:

- Predefined routines "built in" to Lightspeed Pascal, including full access to the Macintosh Toolbox.
- The ability to divide a program into separately compilable units, which supports the development of large programs.
- Numerous standard Macintosh libraries that provide routines for printing, communications, 3-D graphics, speech, and numeric processing.
- Access to other previously-written libraries.
- Access to externally-written assembly language routines and use of in-line machine code.

With the exception of the last point (which is discussed in Chapter 11, "Interfacing with Assembly Language"), these topics are covered in this chapter.

Predefined Procedures and Functions

Lightspeed Pascal provides a large number of predefined procedures and functions. All of the standard Pascal predefines are available. In addition, routines have been added to facilitate address arithmetic, string manipulation, event handling, I/O, and other common tasks. On top of this, nearly all of the procedures and functions (as well as constants and types) described in *Inside Macintosh* are supplied as predefines. As a result, you can easily create Macintosh applications without having extensive knowledge of the Macintosh.

Some of the Macintosh routines are supplied as units; see the section "Standard Libraries" below.

For a complete description of the Lightspeed Pascal predefines, see Sections 9 and 10 of the Language Reference and Appendix E.

All of the special features of the Macintosh—windows, menus, event-handling, sound, and so on—are made easily available for use by your programs by the routines that make up the "Macintosh Toolbox." Many of these routines are stored directly in the Macintosh's ROM (Read-only memory); others are stored in the MacTraps library.

However, you can use them without knowing where they are, because all of the basic Toolbox types, constants, variables, procedures, and functions (those defined in the MemTypes, QuickDraw, OSIntf, ToolIntf, and PackIntf units of Lisa Pascal) are predefined names in Lightspeed Pascal. The MacTraps library will automatically be loaded when you build a program that uses any of the routines it includes.

In addition, the DrawLine, DrawCircle, and PaintCircle routines of Macintosh Pascal are supported, as are the synonymous parameter lists for the Quickdraw Rect, RoundRect, Oval, and Arc routines. Thus,

```
var
    r:rect;
begin
    SetRect (r, left, top, bottom, right);
    FillRect (r, dkGray);
```

can be replaced by the more concise:

```
FillRect (top, left, bottom, right, dkGray);
```

Using Units

A Lightspeed Pascal program may be composed of several files, one of which is the usual main program, and the rest of which are called *units*. The use of units is important for two reasons:

- They allow you to break a large program into manageable pieces. You can therefore write arbitrarily large programs.
- They improve turnaround time when changes are made, due to Lightspeed Pascal's ultra-fast linker and its ability to track and update only those files that have been edited.

The use of units is also consistent with Pascal's emphasis on structured programming.

A unit consists of an *interface*, which describes how to use the data types, variables, procedures, or functions in the unit, and an *implementation*, which contains the Pascal statements that are the actual executable code. An example of a unit is shown below.

```
unit Calculations;      { Sample Unit - doesn't do much }

interface

    uses
        Sane;           { This unit needs the SANE library }
```

```

var

    DisplayWidth : integer; { This variable is exported }

{ Procedures & functions: note that parameters are declared }
{ here, and not in the actual code in the implementation part}

procedure Initialize (Precision: RoundPre);

function Square (aNumber: longint) : longint;

implementation

    const

        ConstantPrivateToCalculations = 42;

    var

        RoundingPrecision: RoundPre;

    procedure Initialize; { Parameter list already declared }
    begin                  { in interface }
        RoundPrecision := Precision;
        SetRound (RoundPrecision);
    end;

    function Square; { Parameter list already declared }
    begin            { in interface }
        Square := aNumber * aNumber;
    end;

    procedure PrivateToCalculations(x: integer);
    begin
    end;

end.

```

All names mentioned in the interface part are exported to files that *use* this unit. Also, procedures and functions in the interface must consist of a heading only. The body is elaborated in the implementation. The body must not contain the redundant procedure and function headings previously declared in the interface.

The **uses** clause specifies the dependencies between files comprising a program. This dependency information is used by Lightspeed Pascal to determine which files are affected by an edit (and which therefore must be recompiled).

When you show the Project Window by build order, all units must be listed in the order in which they are used. In other words, files that are used must precede the files that use them, with the

main program always last. See Chapter 5 for details on how to change the build order. See Section 8 of the *Language Reference* for more information on units.

A unit may not exceed about 2500 lines, including the interface parts of any units which it uses. There is therefore a strong incentive to keep interfaces small. (However, you don't need to worry about removing comments or spaces from interfaces, since Lightspeed Pascal doesn't include them when compiling a file.) In particular, if a unit A uses unit B, and only the implementation part of B actually uses anything from unit C, then A need not use C, thus increasing the maximum number of lines available to A.

Standard Libraries

A number of standard Macintosh libraries are provided.

Each library is supplied as two files: an interface (source) file, and a library (object code) file. Both of these files must be added to the project in order to use a particular library.

To reference the definitions in the interface, add the library's unit name to a **uses** clause in your program. For example:

```
uses
    SANE, Graf3D, SpeechIntf;
```

Be sure to place the interface file in the correct build order in your project.

The following libraries are available:

<u>Description</u>	<u>Interface File</u>	<u>Library File</u>	<u>Unit Name</u>
SANE (Floating-Point Math)	SANE	SANELib	SANE
Print Manager	MacPrint	PrLink	MacPrint
Fixed-Point Math	FixMath	FixMathLib	FixMath
3D Graphics	Graf3D	Graf3DLib	Graf3D
Appletalk Manager	AppleTalk	ABPasCalls	AppleTalk
MacinTalk (Speech)	Speech	SpeechLib	SpeechIntf

The interfaces to all of the routines in these libraries are listed in Appendix E.

Also provided with Lightspeed Pascal are the MacinTalk driver (Macintalk), the RAM Serial Driver (SERD), and the Appletalk ATPL package (ABPackage). These are resource files that must be merged with your project's resource file. See Chapter 10 for information on merging resource files; see the appropriate Apple documentation for information on each of these drivers.

The EXTERNAL Directive

The *external* directive is supported. It can be used in place of a procedure/function body to indicate linkage to a separately compiled or assembled routine.

In a unit, procedures and functions declared in the interface part may later be defined as external in the implementation part.

The *external* directive is normally used to refer to assembly language routines. See Chapter 11 for more information on interfacing with assembly language.

The routine that the *external* directive refers to may be written in Pascal. If so, it must be in a separate unit, and that unit must not be *used* by the file where the *external* directive appears.

Chapter 9

Building Applications and Libraries

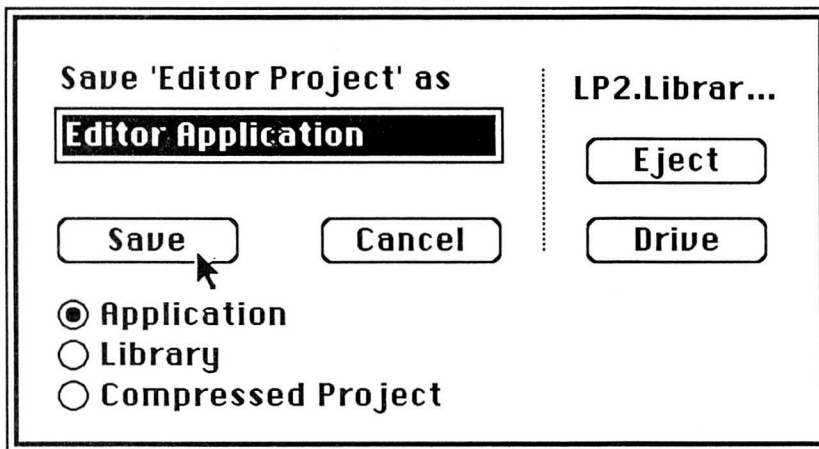
A program developed with Lightspeed Pascal can be saved as either a Macintosh Application program or a Library.

An application program is a program that can be executed independently from the Lightspeed Pascal environment. An application is launchable by double-clicking on its icon on the desktop.

A library is simply the saved object code of one or more (usually many) units. The advantage of libraries is that the source code is not part of the library, only the object code. But this object code can still be used by programs in the Lightspeed Pascal environment. All that users need is the library code file and, optionally, its Pascal interface file.

Applications

To save a project as an application choose **Build & Save As...** from the **Project** Menu. A dialog box like this will appear:



Click on the **Application** button. The default name for the application is "xxx Application", where "xxx" is the name of the project truncated at the first blank from the end. (For example, the default name for "Editor Project" is "Editor Application"). Use the default name or type in another name for the application and then press the **Save** button.

A program is usually saved as an application when its development is complete and it needs to be packaged for distribution. An application program contains only the executable code for the program. None of the Lightspeed Pascal debugging features (Instant, Observe, stepping, etc.) are available.

There are a couple of minor changes you may need to make to your code before saving it as an application. These changes are described below.

Toolbox Initialization

Lightspeed Pascal automatically initializes certain aspects of the Macintosh Toolbox both when the program is run as a Lightspeed Pascal project and when it is saved as an application. The initialization is equivalent to the following calls:

```
InitGraf(@ThePort);
InitFonts;
InitWindows;
InitMenus;
TEInit;
InitDialogs(nil);
InitCursor;
SetApplLimit(<current value of A7> - <Run Options Stack Size>)
MaxApplZone;
```

If you previously used Macintosh Pascal, you are probably unaware that the Toolbox was being initialized for you. If you are porting Macintosh programs originally written for another Pascal compiler, you probably made these calls explicitly in your program. While you are running your program as a Lightspeed Pascal Project, this "double initialization" won't cause any problems.

When you save the project as an application, Lightspeed Pascal will no longer be able to protect you. If the initialization statements are already present in your code, you can either remove them, or disable Lightspeed Pascal's automatic initialization by using the *\$I* compiler directive in the file containing your main program. This directive is ignored in units.

Insert the following line in your program immediately following the *program* statement.

```
{$I-}
```

If a program contains a *\$I-* compiler option then the initialization will not be performed. The default setting is *\$I+*.

Lightspeed Pascal Window Initialization

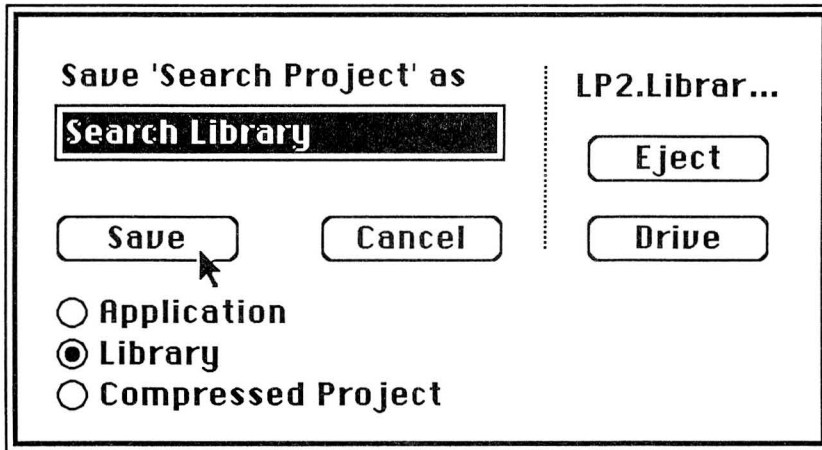
Lightspeed Pascal allows you to open the Text and Drawing Windows from the **Windows** Menu. If a program uses these windows, then certain initialization calls may need to be added before the program is saved as an application. If these calls are not performed, there will be no default windows opened for text and drawing. These calls are:

```
ShowText;
ShowDrawing;
```

You may also want to set the Text and Drawing Window size and placement. You can do this with *SetTextRect* and *SetDrawingRect*. See Section 10 of the *Language Reference* of this manual for additional information on these calls.

Libraries

To save a project as a library choose **Build & Save As...** from the **Project** Menu. A dialog box like this will appear:



Click on the **Library** button. The default name for the library is "xxx Library", where "xxx" is the name of the project truncated at the first blank from the end (e.g. The default name for "Search Project" is "Search Library"). Use this default name or type in another name for the library and then press the **Save** button.

A project can be saved as a library only if the following conditions are met:

- 1) There must be only one code segment in the project.
- 2) There must be at least one file in it.
- 3) There must be no main program (only units).

Before saving as library, you must remove the main program (if there is one). You should also remove the predefined libraries *MacPasLib* and *MacTraps*. Otherwise, when you add the library to a project, the *MacPasLib* and *MacTraps* symbols will be multiply defined, if that project has already used them.

Calling Libraries via Pascal Interfaces

When a project is saved as a library, only the compiled code is saved. To call routines in a

library, you can create a Pascal interface to the library. This can be done quite easily. Declare all routines you wish to be visible to users of the library in the interface. Then in the implementation part declare all those routines as external. For example:

```
unit MyLib;  
interface  
  
    procedure Procl (i: integer);  
    function Func1: boolean;  
  
implementation  
  
    procedure Procl;  
    external;  
  
    function Func1;  
    external;  
  
end.
```

Saved libraries may contain variables. Variables declared in the interface parts of any units in the library are "global" to the linker, and as such may be referenced from assembly language. Note however that there is no way to directly reference such variables from Pascal, since there is no analog to the *external* declaration for variables.

Calling Libraries via External Declarations

You don't necessarily need to create a Pascal interface in order to call a library. You can also call library routines by inserting the external declarations for the routines you want in your program. For example:

```
program foo;  
  
    procedure Procl (i: integer);  
    external;  
  
    function Func1: boolean;  
    external;  
  
begin  
    ...  
end.
```

When you are only calling a few library routines from a single program it may be easier to use external declarations than to create an interface for the library. But when you are calling many library routines from a number of different files, creating an interface may be more convenient and less prone to error than repeating the external declarations in all the programs.

Warning: Lightspeed Pascal does not check that the declarations listed in the library interface or external declarations match the actual library routine declarations. If the declarations do not match, your program will probably not behave as expected. It is the programmer's

responsibility to make sure the declarations exactly match. Lightspeed Pascal only links the names to the actual code.

Compiler Options in Applications and Libraries

While we have not yet discussed compiler options in detail (they're covered in Chapter 13), you may need to know a bit about them before you save your code as an application or library. If you want more information, feel free to jump ahead before reading this section.

Debug

When a program is saved as an application or a library, the debug code (that code controlled by the `$D` compiler directive and the D box in the Project Window) is automatically removed. Lightspeed Pascal does this by recompiling those units which had debug code enabled as if the debug option was disabled. Debug code is not needed for a saved application because the application is not executed under the Lightspeed Pascal environment. Nor is it needed for a library because the source files for the code are not in the library.

Range Checking and Overflow Checking

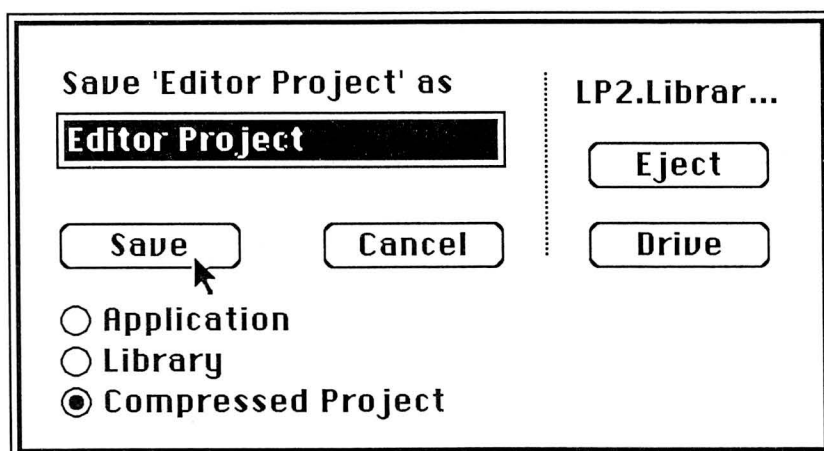
The code generated to do range checking and overflow checking (that code controlled by `$R` and `$V` directives and the R and V box in the Project Window, respectively) is not automatically removed when a program is saved as an application or a library. But it may be a good idea to explicitly turn off these options before saving as an application or a library. When an application is executed and the range or overflow checking code detects an error, the application will drop into Macsbug, or will bomb if Macsbug is not installed.

Names

The name option (controlled by the `$N` directive and the N box in the Project Window), which controls the insertion of procedure and function names into the code is also not automatically removed when a program is saved as an application or a library. If the application or library might be debugged with Macsbug, it is probably a good idea to have the names in the code. If the application or library will not be debugged with Macsbug or if there is a reason not to have the names available, then the name option should be disabled when the program is saved as an application or a library.

Compressed Project

A compressed project is simply a project with all compiled code removed from the project document. The only reason for saving a project as a compressed project is for space savings on a disk. When you are making backups or when a disk is getting full you may want to save as compressed project. When you try to execute a compressed project, the entire project will need to be rebuilt. To save a project as compressed project choose **Build & Save As...** from the **Project** Menu. A dialog box like this will appear:



Click on the **Compressed Project** button. The default name for the compressed project is same as the current project. Use the default name or type in another name for the project and then press the **Save** button.

Smart Linking

When you **Build & Save As...** an application, the "smart" linker determines which files (units and/or libraries) contain routines which are reachable on a reference chain starting from the main program. Unreachable files are removed; they are not actually removed from the project, but the code and data that they contribute will not appear in the application.

If a library file is composed of many smaller files, the code and data contributions of each of the component files are remembered. When a project containing such a library is built into an application, the smart linker will actually remove unused components of the library while retaining necessary ones.

The standard libraries *MacPasLib* and *MacTraps* were created in this fashion. They are actually composed of a large number of smaller libraries, each of which was created using the MDS assembler and the .Rel Converter (see Chapter 11). Therefore, although the libraries appear to be large in the Project Window, typically only a small subset of the total code is present in your saved applications.

Chapter 10

Using Resources

A typical Macintosh application uses *resources* to define its menus, windows, dialogs, alerts, strings, pictures, and icons. In addition, the application's code itself consists of resources. These objects typically reside together in a single resource file. This chapter tells you a few special things you need to know to use resources with Lightspeed Pascal.

(If you are new to programming the Macintosh, it is important to learn about resources right away, as they pervade every aspect of the Macintosh Toolbox. Refer to *Inside Macintosh* or *Macintosh Revealed* for detailed information.)

Using a Separate Resource File

Lightspeed Pascal allows you to run a program without first building an application file. That, of course, is part of why it is so fast. But if there is no application file, where do your program's resources come from?

The answer is that they come from a separate resource file. (Later in this chapter, we'll show an example of what a resource file looks like, if you are not already familiar with them.)

Lightspeed Pascal allows you to designate one resource file to be used by your program. Choose **Run Options...** from the **Project** Menu. A dialog box like this will appear:

Run-time Environment Settings

Resources ☐ Use resource file: for resources used by the project.

Text Window Text Window saves characters

☐ Echo to the printer

☐ Echo to the file:

Monaco 9

Memory Stack size: kilobytes

Zone size: kilobytes

Check the "Use resource file:" check box and choose a resource file to use. This resource file will be opened automatically when your program is executed.

Note: The resource file is required to be on the same volume (and in the same folder if using HFS) as the project.

Combining Resource Files

When you are ready to prepare a stand-alone version of your program—that is, one that can be launched from the Finder rather than only from Lightspeed Pascal—choose **Build & Save As...** from the **Project** Menu. Then click on **Application**, give the application a name, and then click on **Save**.

This procedure creates a resource file containing the code resources that constitute your application. In addition, Lightspeed Pascal copies to that file all the resources from the separate resource file designated by the **Run Options...** command (see above). The result is a stand-alone application that contains everything it needs to be launched from the Finder.

An Example

Here is a sample program, written without using resources, that just opens a window and waits for you to click its "go-away" box.

```
program WithoutResources;
var
    MyWindow, WhichWindow : WindowPtr;
    MyEvent : EventRecord;
    Bounds : Rect;
    Done : Boolean;
begin
    SetRect(Bounds, 100, 100, 300, 300);
    Done := false;
    MyWindow := NewWindow(nil, Bounds, 'Sample Window',
                          true, 0, nil, true, 0);
    repeat
        if GetNextEvent(EveryEvent, MyEvent) then
            if MyEvent.What =MouseDown then
                if FindWindow(MyEvent.Where, WhichWindow) =
                    InGoAway then
                    if WhichWindow = MyWindow then
                        if TrackGoAway(MyWindow, MyEvent.Where) then
                            Done := true;
    until Done;
end.
```

Here is the same program written using resources:

```
program WithResources;
var
    MyWindow, WhichWindow : WindowPtr;
    MyEvent : EventRecord;
    Done : Boolean;
begin
    Done := false;
    MyWindow := GetNewWindow(128, nil, nil);
    repeat
        if GetNextEvent(EveryEvent, MyEvent) then
            if MyEvent.What =MouseDown then
                if FindWindow(MyEvent.Where, WhichWindow) =
                    InGoAway then
                    if WhichWindow = MyWindow then
                        if TrackGoAway(MyWindow, MyEvent.Where) then
                            Done := true;
    until Done;
end.
```

What's In A Resource File?

Resource files can be created with either the RMaker or ResEdit utility from Apple. Both of these utilities are supplied with Lightspeed Pascal. RMaker is described in Appendix H. ResEdit is described in Appendix I.

Here is an RMaker script for a sample resource file *Sample.Rsrc*. You can use this script to create a resource file for use with the previous example.

```
Sample.Rsrc                ;; output file
????????                ;; file type and creator

Type WIND
    ,128                    ;; resource ID
Sample Window
100 100 300 300            ;; window bounds
Visible GoAway
0                          ;; a regular document window
0                          ;; the refCon (not used in the
                          ;; sample program)
```

You can create a text file containing this script using MacWrite or any other text editor (such as the text editor demo included with Lightspeed Pascal). If you use MacWrite, be sure to save your file as Text Only. Call the file *Sample.R*. Then run RMaker on this file. This procedure will create a resource file called *Sample.Rsrc*.

Alternatively, you can use ResEdit to create the file. ResEdit provides an interactive interface for building resources. See Appendix I for details.

Once the resource file is created, it needs to be designated as the file containing the resources used by your project. Choose **Run Options...** from the **Project** Menu. Check the "Use resource file:" box and choose *Sample.Rsrc*. This file will be opened automatically when your program is executed.

Chapter 11

Interfacing with Assembly Language

This chapter contains the information you need to interface to assembly language, and to understand how your Pascal programs communicate with the Macintosh hardware and software environment. In addition, an understanding of this material is a necessary prelude to using LightsBug or Macsbug for low-level debugging of your programs. However, while this chapter does make an effort to review some fundamental concepts of the Macintosh architecture, it is nonetheless no substitute for the complete treatment given in *Inside Macintosh* (Addison-Wesley, 1985).

Pascal is the Macintosh's "native" language. The Macintosh system software expects to be called from Pascal, and the internal Macintosh data structures usually correspond to Pascal data types such as pointers, records, and arrays.

Of course, down deep the real native language of the Macintosh is the MC68000 assembly language. This chapter describes the Pascal procedure calling conventions at the machine level, and explains how you can incorporate assembly language code into a Lightspeed Pascal project using the MDS (Macintosh Development System) assembler. A detailed description of the internal representation of the Pascal data types is also provided to help you in understanding and debugging your programs at the assembly language level.

Run-Time Architecture

The run-time environment of your Macintosh application has three fundamental components: the *stack*, the *heap*, and the *A5 world*.

The Stack

The stack is an area of memory that is dynamically allocated and deallocated in a strict last-in-first-out fashion, like a stack of trays in a cafeteria. The MC68000's A7 register is reserved as a *stack pointer*: it always contains the address of the top of the stack. (Note that the stack of the MC68000 actually grows downward, towards lower memory addresses, so that the top of the stack is actually the byte in the stack with the lowest address.)

The stack contains information pertaining to the activation and deactivation of procedure and functions. Each time a routine is called, a *stack frame* is allocated. This stack frame contains all of the routine's parameters, local variables and temporaries, and the return address. When the routine exits, the stack frame is released and the context of the calling routine is restored. The processor's A6 register is used as the frame pointer of the currently active procedure or function.

The Heap

The heap (usually called a *zone* in Macintosh terminology) is an area of memory from which arbitrary-sized pieces can be allocated and deallocated in any order. All of the dynamic memory required by your program is allocated from the heap: windows, menus, code segments, resources, even other heap zones. The various "housekeeping" tasks necessary for the random allocation and deallocation of memory are performed by a part of the Macintosh operating system called the Memory Manager.

The memory within a zone is divided up into contiguous pieces called *blocks*, of which there are three types:

- Free Blocks - these blocks represent unused heap memory that may be allocated to satisfy a memory request.
- Nonrelocatable Blocks - these are allocated blocks of memory that reside at a fixed location. They are referenced by a *pointer* to the block.
- Relocatable Blocks - these are blocks that may be moved by the Memory Manager, usually to gather sufficient contiguous free space to satisfy an allocation request. Because a relocatable block may be moved, your application cannot keep a pointer to it. Instead, a nonrelocatable *master pointer* to the relocatable block is maintained by the Memory Manager. Your program accesses the block via a pointer to this master pointer, called a *handle*, which must be dereferenced, not once, but twice to access the data.

The Memory Manager is one of the most fundamental and pervasive parts of the Macintosh operating system. There is a great deal of lore concerning its effective use, far more than can be covered here. For more complete information, refer to the Memory Manager section of *Inside Macintosh*.

The A5 World

The A5 world is a colloquial term for the area of memory that is referenced via the processor's A5 register. This is a distinct area of memory, which has three components:

- Application Parameters - the first 32 bytes of memory "above A5" contain the Finder's *startup handle* at 16(A5) and the all-important pointer to the Quickdraw globals at 0(A5). The remaining 24 bytes are unused.
- Jump Table - the remainder of the "above A5" area contains the Jump Table, a data structure used by the Segment Loader to manage the loading and unloading of code segments. Every procedure and function that is referenced across segments has an entry in this table: in the unloaded state the entry contains instructions that cause the segment containing the routine to be loaded, and in the loaded state it contains an absolute jump to the routine.
- Application Globals - the "below A5" area of memory contains the application's global variables, including the Quickdraw globals.

Pascal Data Types

Integer Types

The integer-type *integer* is represented as a 16-bit two's complement number with a range of -32767 to 32767. The integer-type *longint* is represented as a 32-bit two's complement number with a range of -2147483647 to 2147483647.

Subranges of integer that are in the range -128 to 127 are represented as an 8-bit number, and thus occupy a single byte. Subranges with upper and/or lower bounds outside this range occupy a word.

There is a special case: if an integer subrange in the range 0 to 255 is a component of a packed structured-type, it is represented as an unsigned byte.

Chars

The type *Char* is represented as a 16-bit value with the extended ASCII code of the character in the low-order byte, and 0 in the high-order byte.

If a *Char* is a component of a packed structured-type, it is represented as an unsigned byte.

Booleans

The type *Boolean* is represented as a byte quantity that may assume only the values 0 (false) or 1 (true).

Enumerated Types

Enumerated types are represented as unsigned byte quantities that may assume ordinal values in the range 0 to 255 (depending upon the number of enumerated constants in the type).

Real Types

The real-types *Real*, *Double*, and *Extended* are represented as IEEE-format floating-point numbers of 32, 64 and 80 bits respectively. The real-type *Computational* is represented as a 64-bit two's complement integer with a unique reserved value.

The floating-point formats are described in complete detail in Appendix D.

Pointers

Pointer-types are represented as 32-bit address values in which only the low-order 24 bits actually participate. Note that in the Macintosh environment the high-order 8 bits are not guaranteed to be 0: this may cause obscure errors in address arithmetic.

Strings

A string-type of size n has a 1 byte length field followed by n bytes containing the character components of the string (each occupying a single byte as if packed). An unused byte is added to the end if needed to ensure that the total number of bytes is even.

Arrays

An array-type with index $[L..H]$ is represented as if $H-L+1$ variables of the component type were laid end to end. If the size of the component type is not 1, it is first rounded to an even number of bytes so that each element of the array is on an even byte boundary. An unused byte is appended to the array if necessary to cause it to occupy an even number of bytes.

A multidimensional array of indices $[L1..H1, L2..H2, \dots, Ln..Hn]$ is represented as if it were declared as an array of index $[L1..H1]$ with a component array of index $[L2..H2]$ with a component array of index $[L3..H3]$, etc.

A packed-array-type is identical to the corresponding array-type unless the component type is packable (i.e. char-type or integer subrange in the range 0..255) in which case the components are allocated in their packed format.

Records

A record with fields $f1: T1, f2: T2, \dots, fn: Tn$ is represented as if each field were a single variable and all fields were laid end to end. If a field's size is not 1, the field is first aligned to an even boundary. An unused byte is appended to the record if necessary to cause it to occupy an even number of bytes.

A packed-record-type is identical to the corresponding record-type unless the types of one or more fields are packable (i.e. char-type or integer subrange in the range 0..255) in which case the fields are allocated in their packed format.

Sets

Sets are represented as bit arrays where each bit indicates whether the corresponding element is "in" the set or not. Sets occupy an even number of words, and are always allocated as if the set origin were 0: a set of 0..255 occupies the same number of bytes as a set of 100..255.

There is a special case: a set whose elements are in the ordinal range 0..7 is stored as a byte. However, when passed as an actual value parameter, the set is first extended to a word, so the value ends up in the low-order byte rather than the high-order byte.

Files

File-types are represented as records that contain 32 bytes of status information, a 522 byte access-path buffer, and a component buffer whose size is equal to the size of the component type of the file.

Files of type *Text* are represented identically to **packed file of Char**. However, the predefined routines *eoln*, *writeln*, and *readln* can only be used with files of type *Text*. See Section 9 of the *Language Reference*.

Pascal Calling Conventions

Calling Sequence

When a Pascal routine is called, it is the caller's responsibility to reserve space for the return value (if the routine is a function), and push the actual parameters onto the stack in order from left to right. The caller's code looks something like this:

```
SUBQ      #n, A7                ; reserve space for return value
MOVE      ..., -(A7)            ; push first argument
...
MOVE      ..., -(A7)            ; push last argument
JSR       routine               ; jump to subroutine
```

The return value (if any) is on top of the stack upon return from the call. If the routine being called is a stack-based Toolbox routine, the appropriate trap instruction is generated in place of the JSR instruction. For register-based Toolbox calls, a JSR to an assembly-language "glue" routine is generated.

If the called routine is declared in a nested scope, the caller must also provide the *static link*: the frame pointer of the most recent activation of the procedure or function that statically nests the called routine.

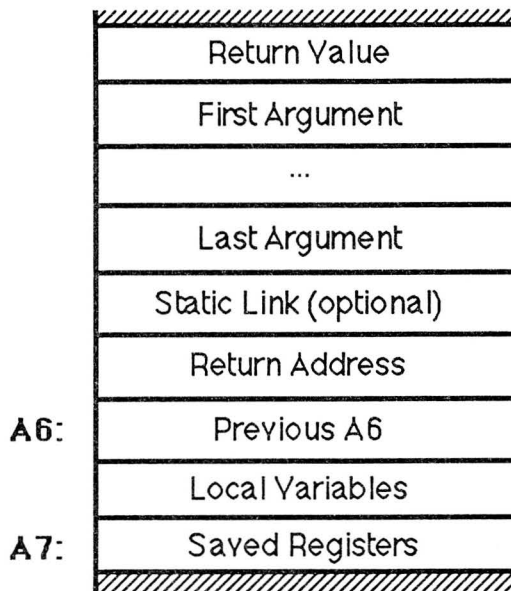
Routine Entry

Upon entry to a Pascal routine, the return address is on the top of the stack. The routine usually creates a stack frame using the LINK instruction, and saves any non-scratch registers that it uses:

```
LINK      A6, #16 bits          ; allocate local variable space
MOVEM.L   ..., -(A7)            ; (optionally) save registers
```

Since the *-n* is a 16 bit value, there is a 32K size limit on the local variable space.

The stack frame has the following format:



The parameters and return value are accessed as positive displacements from A6, while the local variables and temporaries are accessed as negative offsets.

Routine Exit

When a routine exits, it is responsible for deallocating the stack frame, and removing any parameters that were pushed by the caller. The epilogue of a routine usually looks like this:

```
UNLK      A6                ; deallocate stack frame
MOVEA.L   (A7)+, A0         ; get return address
ADD       #n, A7            ; discard parameters
JMP       (A0)              ; return to caller
```

Parameter Passing

The conventions for parameter passing in Pascal are dependent upon the kind of parameter that is passed:

- VAR Parameters - the 32-bit address of the actual parameter is placed on the stack.
- Value Parameters - the value of the actual parameter is placed on the stack, if it occupies 4 bytes or less. Otherwise, the address of the actual parameter is passed, and the caller is required to make a local copy of it (in case it is modified).

All values occupy either 2 or 4 bytes on the stack. A byte value appears in the high byte of the word and is found at an even offset from A7 (or A6).

- VAR STRINGs - the 32-bit address of the string is passed along with a 16-bit size that is the declared size of the actual parameter.

- Procedures/Functions - the 32-bit (jump table) address of the procedure or function is passed, along with the 32-bit static link to be used when the routine is actually called (if the procedure or function is declared in the outermost scope, 0 is passed).

Return Values

If the return value of a function occupies 4 bytes or less, the caller allocates 2 or 4 bytes on the stack for the return value. This value is left on the stack upon return from the function.

If the value occupies more than 4 bytes, a temporary variable of the same size is allocated, and the caller pushes the address of this temporary as a "hidden parameter". The caller discards this address when the called routine returns.

Interfacing with Assembly Language

Assembly language source code can be assembled using the MDS assembler, converted to a Lightspeed Pascal library, and loaded into a project.

Accessing Global Symbols

Procedure and function names defined in assembly language and made public via the XDEF directive may be called from Lightspeed Pascal, as long as they conform to the conventions described previously. Simply create an external declaration for the name:

```
procedure ClearHandle (aHandle: Handle);
external;
```

It is up to you to ensure that the Pascal declaration of the assembly language routine matches the implementation with respect to argument types, value/reference passing, and return values.

Procedure and function names defined in the interface-part of a Pascal unit may be imported to an assembly language source file using the XREF directive:

```
XREF      aPascalRoutine      ; defined in a Pascal unit

...
JSR      aPascalRoutine      ; call Pascal routine
...      ; pop return value, etc.
```

The name of the routine in a call must not be suffixed with the (A5) addressing base; the linker will resolve the reference to a PC-relative or A5-relative jump-to-subroutine as necessary.

Variable names defined in the interface-part of a Pascal unit may also be imported using the XREF directive:

```
XREF      aPascalVar          ; defined in a Pascal unit

...
MOVE.L    aPascalVar(A5),D0    ; load Pascal variable
...
```

In this case, the (A5) addressing base must be supplied, just as it must for names defined in assembly language using the DS directive.

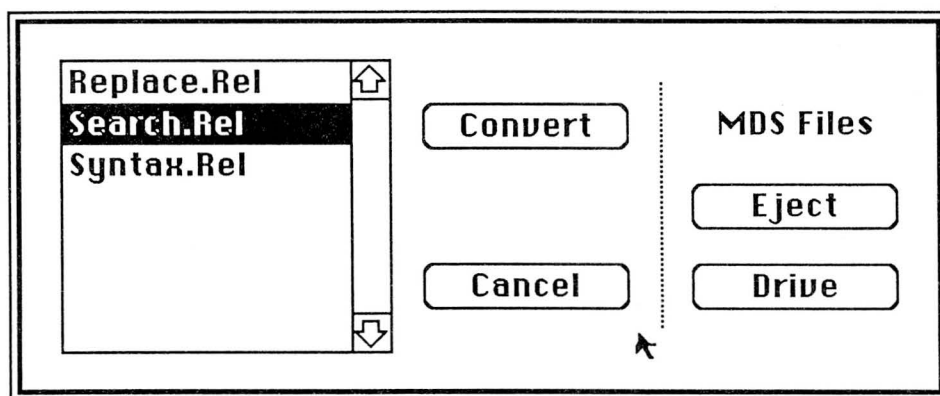
Register Saving Conventions

Data registers D0-D2 and address registers A0-A1 are scratch registers, and may be freely used by any assembly language routine. Other registers must be saved and restored if they are going to be modified. You should avoid using address registers A5-A7.

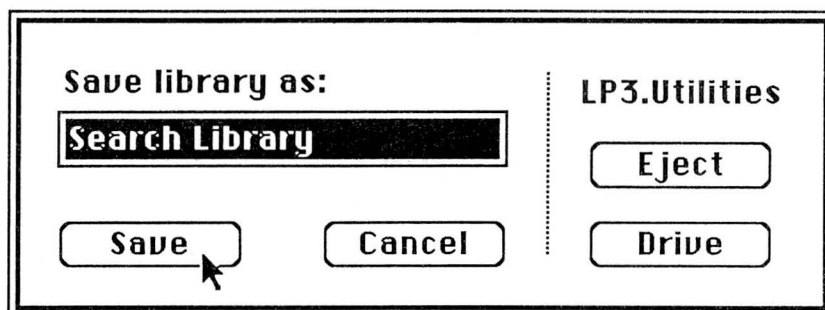
Converting .REL Files

The MDS assembler translates assembly language source files to *.Rel* object files. A *.Rel* file must be converted to a Lightspeed Pascal library before it can be added to a project. The .REL Converter utility is provided for this purpose.

The .REL Converter presents a dialog box allowing you to specify a *.Rel* file to be converted:



When you select one, it presents another standard file dialog that enables you to name the created library file:



Note that the default name of a converted library is the name of the *.Rel* file with the *.Rel* suffix removed, followed by a single space and the word "Library".

The standard file dialogs are presented repeatedly until you click the **Cancel** button, allowing multiple files to be rapidly converted.

Chapter 12

LightsBug—Debugging at Lightspeed

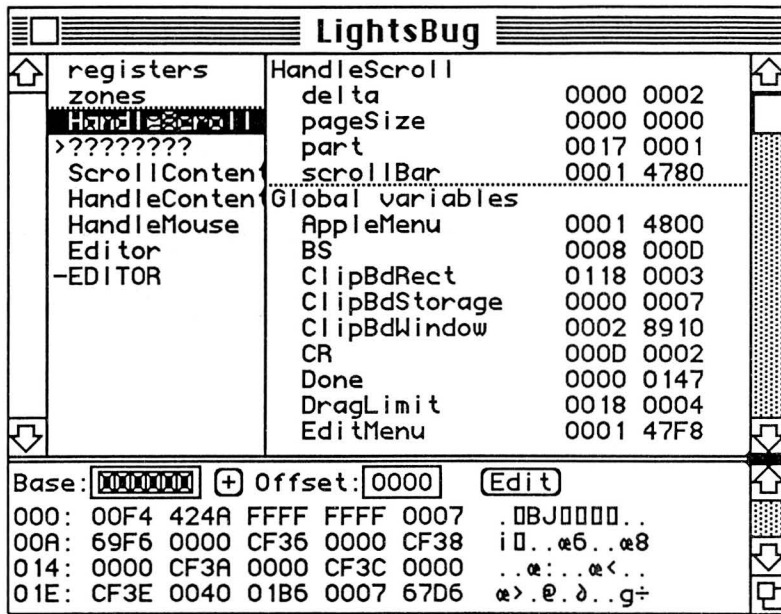
This chapter describes how to debug programs with LightsBug, Lightspeed Pascal's powerful memory level debugger. LightsBug provides the following debugging features:

- A subroutine call chain which displays the names and the activation order of all currently active subroutines.
- A variable display which shows names and values of all variables in a running program.
- A register display which shows the values of the MC68000 registers while running a program.
- A zone display which shows all objects in the system and application zone.
- A memory display which allows memory to be displayed and modified.
- Support for breakpoints at Toolbox calls.
- Support for the examination of compiled code.

To enter LightsBug, choose **LightsBug** from the **Windows** Menu. The LightsBug Window is divided into three panes or parts. The upper left pane displays the subroutine call chain and also controls the contents of the Register/Zone/Variable display. The upper right pane is the Register/Zone/Variable display. The lower pane is the memory display.

Subroutine Call Chain

The upper left pane of the LightsBug Window displays the subroutine call chain in the area below the dotted line. The subroutine call chain is the list of all the subroutines which are currently activated (i.e. all subroutines which have been called, but have not as yet finished execution). The subroutines are listed in reverse order of being called (i.e. the last subroutine called is at the top of the list). The very last name is the name of the main program. For example:



The spelling of the subroutine names displayed in the call chain will either be:

- 1) Exactly as it is spelled in your program—If the subroutine is contained in an open editing window and the routine has been compiled with the Debug option on.
- 2) Truncated to eight characters, uppercased, and underscores removed—If the subroutine is not contained in an open editing window or the routine has been compiled with the Debug option off, but the routine has been compiled with the Names option on.
- 3) ??????????—If the subroutine is not contained in an open editing window or the routine has been compiled with the Debug option off, but the routine has been compiled with the Names option off.

The prefixes on the subroutine names in the subroutine call chain indicate the following:

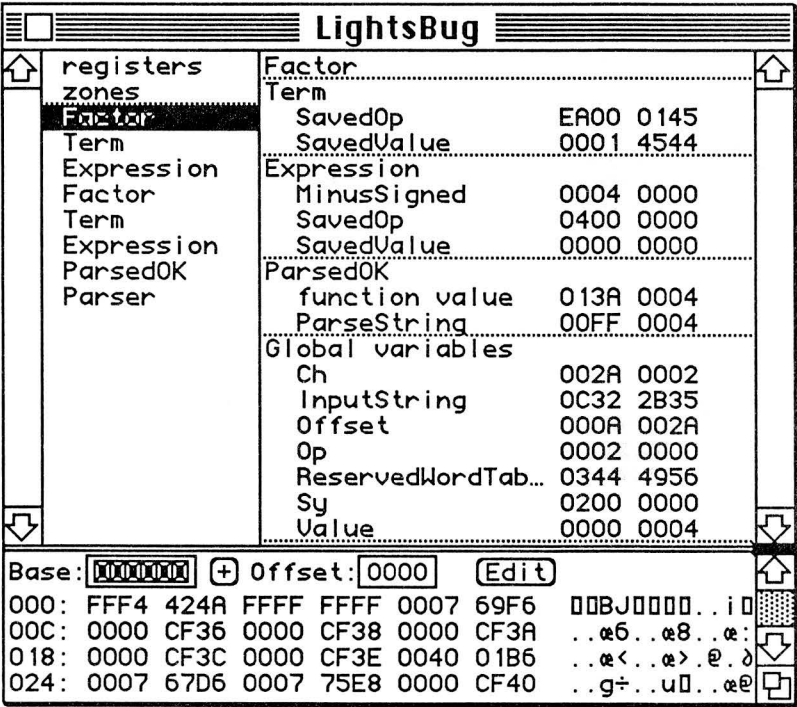
- ' ' This subroutine is contained in an open editing window. This subroutine has been compiled with the Debug option on.
- '-' This subroutine is not contained in an open editing window. This subroutine has been compiled with the Debug option on.
- '>' This subroutine has been compiled with the Debug option off.
- '*' A gap in the call chain was detected. This could be caused by an assembly language routine using register A6 in an unusual way.

In the example just shown, *EDITOR* is the program name. *EDITOR* called a procedure *Editor* which called *HandleMouse* which called *HandleContent* which called *ScrollContent*. *ScrollContent* called a Toolbox routine which is listed as ??????????. The toolbox routine in turn called *HandleScroll*. Notice that *EDITOR* is marked as not being contained in an

open editing window. ???????? is marked as having been compiled with the Debug option off.

Variables

Clicking on any subroutine name in the subroutine call chain will change the display of the upper right pane of the LightsBug window. The names and the hexadecimal values of the variables of that subroutine will be displayed. Also displayed will be the names and values of the variables of all subroutines within which the selected subroutine is statically nested within. This means that all variables visible (according to the scope rules of Pascal) to the selected subroutine will be displayed. An example LightsBug Window looks like this:



A subroutine name and its variables and parameters are separated from other subroutines by a dotted line. The subroutine name is listed first, followed by the current function value (if appropriate), and the names of its variables and parameters (if any). Beside the name of each variable or parameter will be its value displayed in hexadecimal. In this example, the subroutine *Factor* was selected. The routines that *Factor* is statically nested within are *Term*, *Expression*, *ParsedOK*, and *Parser*. *Factor* has no local variables or parameters. *Term* has two local variables—*SavedOp* and *SavedValue*. *Expression* has three local variables—*MinusSigned*, *SavedOp*, and *SavedValue*. The function *ParsedOK* has a function value and one parameter—*ParseString*, which is passed by reference (i.e. the value displayed is a pointer to the parameter, not the actual value). The global variables of *Parser* as well as their values are listed last under the heading "Global variables".

Not all subroutines in the call chain may be in the variable display. This is because the selected subroutine may not be statically nested within all currently activated subroutines.

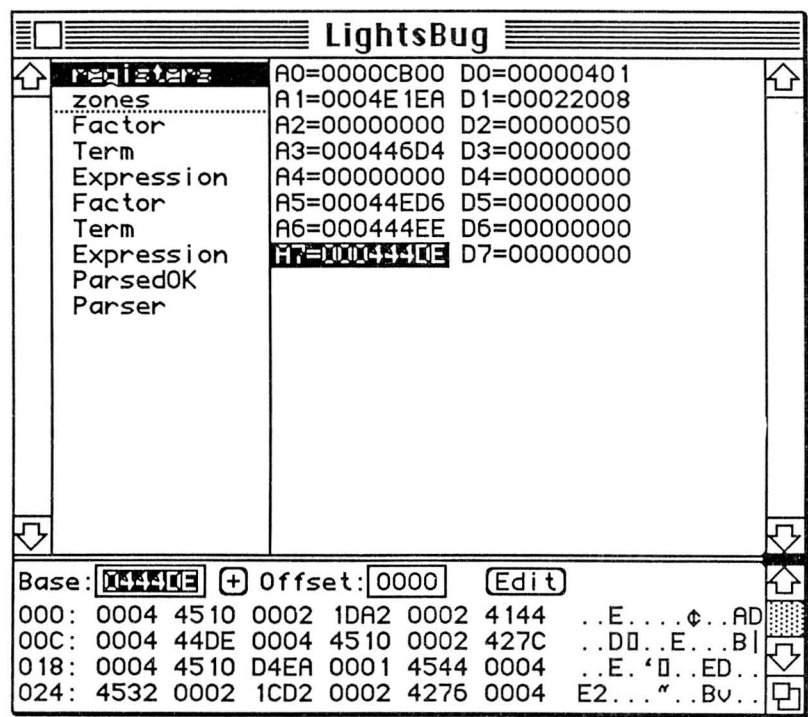
Only global variables visible to the selected subroutine will be shown (i.e. global variables in units that are not known to the selected subroutine are not displayed).

Some subroutines may be listed multiple times in the subroutine call chain. This happens when subroutines are called recursively. For example, *Factor* is listed twice in the subroutine call chain just shown. This indicates that *Factor* was called recursively. Selecting the first *Factor* in the list will display the values of the variables and parameters of the latest activation of *Factor*.

Note: See the example later in this chapter for an example of tracking down a bug in a recursive program.

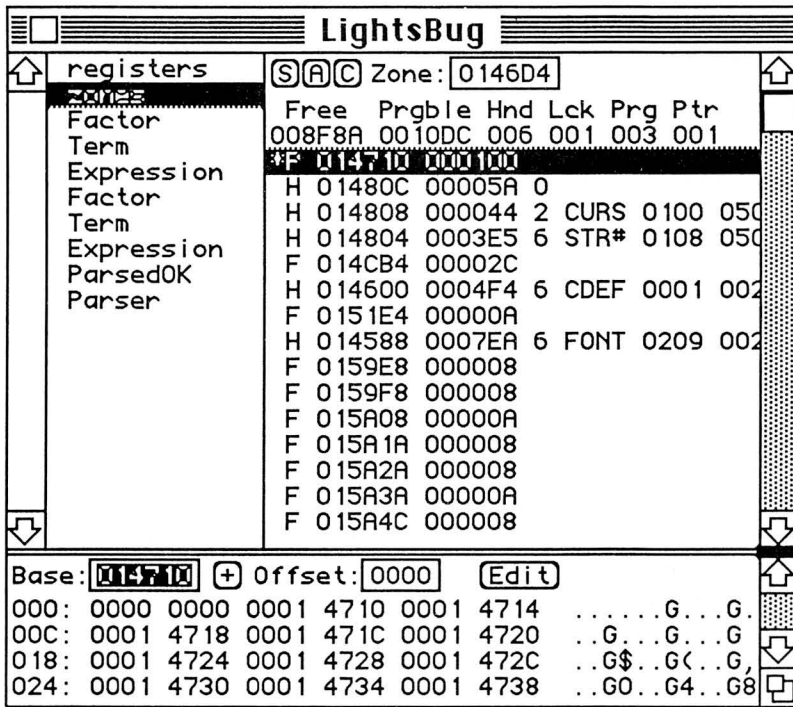
Registers

To display the current value of the MC68000 registers, click on **registers** in the upper left pane. The value of the registers will be displayed in the upper right pane. It will look similar to this:



Zones

To view a zone (also known as a heap), click on **zones** in the upper left pane. The upper right pane will look similar to this:



The zone mode display has additional control buttons labeled S, A, and C:

- S Click on the button labeled S to display the system zone.
- A Click on the button labeled A to display your application's zone.
- C Click on the button labeled C to display your current zone.

Any other zone may be selected by typing its address in the box next to the S A C buttons.

The current zone will be the same as your application zone unless you have done a SetZone.

A summary of the total size and number of each type of object is shown near the top of the pane. The meaning of this summary is:

- Free The total size of all free objects.
- Prgble The total size of all purgeable objects.
- Hnd The number of handled objects.
- Lck The number of locked objects.
- Prg The number of purgeable objects.
- Ptr The number of pointer objects.

The blocks in the zone are displayed in physical order. The format of the zone display, from left to right, is:

- 1 character '*' for locked object or ' ' for unlocked object.
- 1 character 'H' 'P' or 'F' for handle, pointer and free block, respectively.

6 characters	Logical base address of the block in hex, (e.g. for handle h, h^; for pointer p, just p).
6 characters	Logical size of the block in hex.
1 character	For handle blocks only, the attribute nibble in hex. 8 = Locked, 4 = Purgeable, 2 = Resource (e.g. E=Locked Purgeable Resource).
4 characters	For resources only, the resource type.
4 characters	For resources only, the resource id.
3 characters	For resources only, the resource file reference number.

In the window just displayed, the selected entry in the zone display is a pointer with a base address of 14710 and a logical size of 100 hex. The third entry in the zone display is a handle with a base address 14808, a logical size of 44, a Resource, a resource of type CURS, a resource id of 0100, and a resource file reference number of 05C.

Memory Display

The bottom pane of the LightsBug Window is the memory display. Memory is displayed in hexadecimal starting at the memory location Base+Offset. An ASCII display of the same memory is displayed in the rightmost part of this pane.

There are a number of different ways you can choose which memory is displayed. They all involve changing the base address and/or the offset. To choose which memory is displayed you can:

- 1) Type in a new base address and/or a new offset, and then hit the Return or Enter key. The ~ key can be used to revert to what was there before you started typing.
- 2) Click on the + button. This adds the offset to the base and changes the base to that new value. The offset will be set to 0.
- 3) Click on any register displayed in the Register/Zone/Variable part of the window. The value of this register becomes the new base address. The offset will be set to 0.
- 4) Click on any zone entry displayed in the Register/Zone/Variable part of the window. If the zone entry is a pointer, then this pointer becomes the new base address. If the zone entry is a handle, then the dereferenced handle becomes the new base address. The offset will be set to 0.

Note that the actual data area of the memory block is displayed. If you used the Macsbug heap dump command, and then displayed memory at the block address given in the heap dump, you would see the eight byte block header followed by the actual data.

- 5) Click on any variable name displayed in the Register/Zone/Variable part of the window. The address of the variable (not the value) becomes the new base address. The offset will be set to 0.

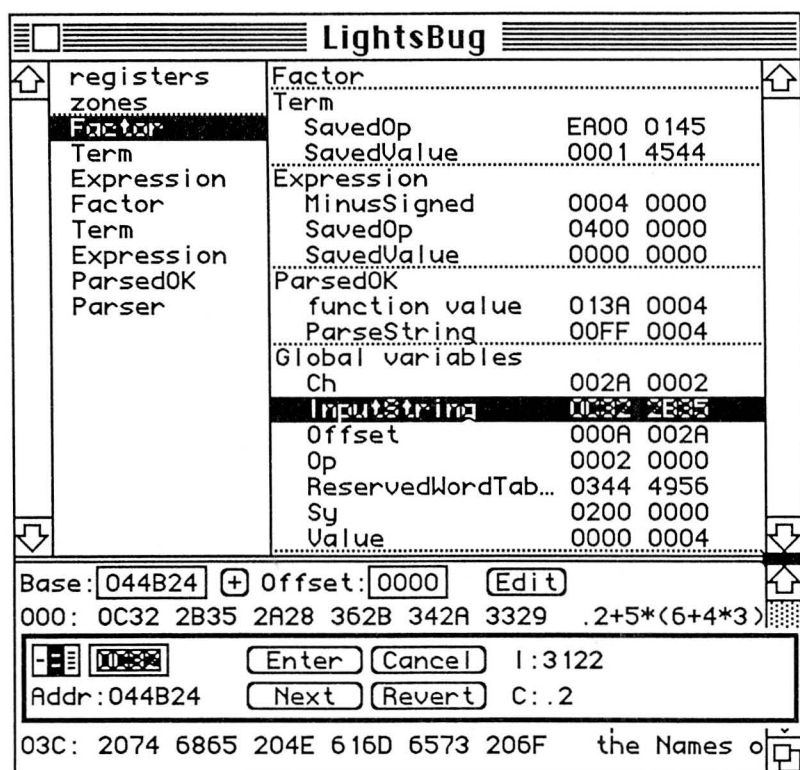
- 6) Click on any subroutine name displayed in the Register/Zone/Variable part of the window. The frame pointer for that activation becomes the new base address. The offset will be set to 0.
- 7) Click on any byte of the memory display. This will cause that byte to become the first byte of the memory display. The offset will be updated accordingly. The base will be unchanged. This is useful for stepping through arrays.
- 8) Shift-Click on any byte of the memory display. The first byte of the memory display will move to the point under the cursor. The offset will be updated accordingly. The base will be unchanged.
- 9) Option-Click or Command-Click on any byte of the memory display. This will cause a dereference of the four bytes starting with the byte selected by the cursor to be the new base address. The offset will be set to 0.
- 10) Option-Command-Click on any byte of the memory display. This will cause a double dereference or handle dereference. The offset will be set to 0.
- 11) The last eight changes to the base address are remembered and may be recalled via the < and > keys (comma and period keys).

Note: The offset field is sign extended (i.e. if you type in a minus sign followed by a hex number, it will compute the proper value).

Editing Memory

The memory displayed in LightsBug can be edited. *Use caution when doing this because careless changes to memory may cause severe problems.*

To edit memory, click on the **Edit** button in the lower pane of the LightsBug Window or hit Shift-E. This will bring up a display similar to this:



The edit box superimposed over the memory display has a number of options. The **Addr** field displays the address of the memory being edited. Initially, this address is the same address as the base address of the memory display. You can edit one, two, or four bytes of memory at a time by clicking on the rectangles in the upper left of the edit box. The rectangles contain one, two, or four lines to indicate the number of bytes being edited. In this example the rectangle with two lines has been selected, and the two bytes of memory starting at **Addr** are displayed.

You edit memory by typing in the box next to the byte number rectangles. When you are done typing you can click on any of the four buttons in the edit box:

- Enter This causes the change to take effect. This changes memory at **Addr** to the value just typed. Both the Enter and Return keys have the same effect.
- Cancel This aborts editing mode. The Shift ~ key has the same effect.
- Next This causes the change to take effect, and displays the next byte, word, or longword of memory. The Tab key has the same effect.
- Revert This restores the edited value to what it was previously. The ~ key has the same effect.

On the right side of the editing box, memory is displayed in different formats, with a letter identifying each format. When the one byte mode is chosen, the memory is displayed as:

- U An unsigned byte.
- S A signed byte.
- C A character.

When the two byte mode is chosen, the memory is displayed as:

- I An integer.
- C Two characters.

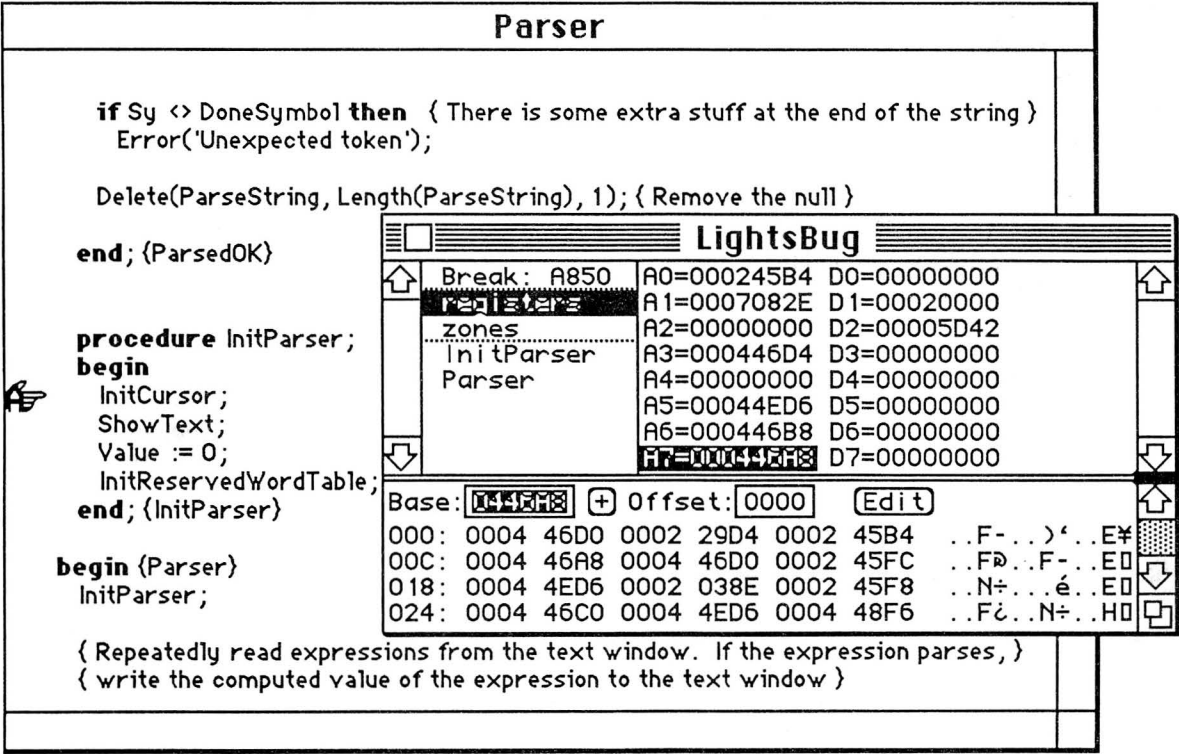
When the four byte mode is chosen, the memory is displayed as:

- L A longint.
- C Four characters.

You can scan through memory in edit mode by hitting the < and > keys. This takes you backwards or forwards through memory by the size of the current mode (e.g. when you are in four byte mode, the < and > keys moves the memory display by four bytes).

A-Traps

LightsBug provides assistance in debugging calls to Toolbox routines. If you want to stop at all Toolbox calls, choose **Break at A-Traps** from the **Debug Menu**. When your program is executing and you hit a Toolbox call, your display will look similar to this:



A special finger (with an A emblazoned on it) will point to the last statement executed which had been compiled with the Debug option on. LightsBug will display the Toolbox Trap number following the word **Break:** in the upper left pane of the LightsBug Window. In this example, the trap number is A850, which is the InitCursor Toolbox Trap.

The Register/Zone/Variable display automatically switches to Register mode with the value of A7 as the base address of the Memory Display. This is helpful because register A7 is used as the

stack pointer, and parameters to most Toolbox routines are passed on the stack. The memory display will therefore show the values of the parameters passed to the Toolbox routine just about to be called.

In this example, register A7 contains the value 446A8, which is also the base address of the memory display. InitGraf takes one 4-byte pointer as a parameter, so the first four bytes of the memory display is that pointer value. The next four bytes are the return address, and so forth.

Note: If you are executing code that was compiled with the Debug option off, the special finger may point to the same statement for many distinct A-Trap breaks.

Examining Compiled Code

A special mode of the **Step** command helps you to use Macsbug to examine compiled code. To examine the code for a specific statement, **Step** your program until the finger is pointing to the statement of interest. Then hold the Shift key down and **Step** again (from the keyboard, hit Fan-Shift-S). This is equivalent to choosing **Step** from the **Run** Menu, except that Lightspeed Pascal will drop you into Macsbug just before the RTS that dispatches to your code.

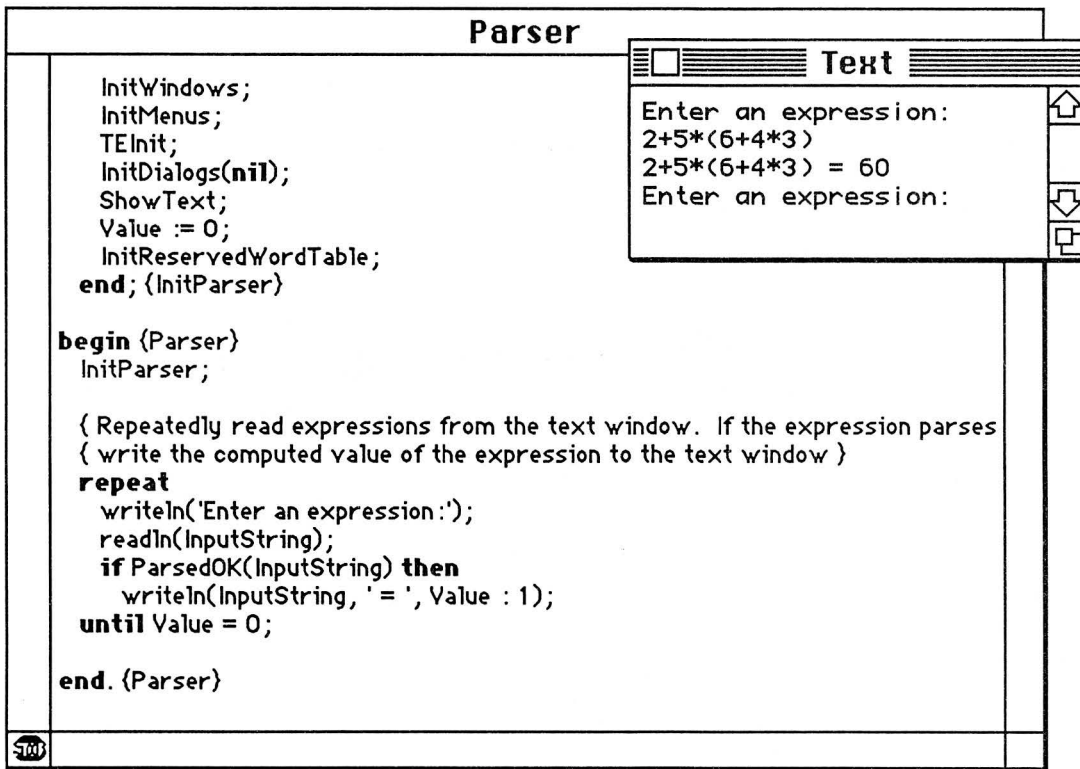
To examine your code, type T, hit Return, then type IL PC and hit Return again. This will cause a disassembly of your code to be displayed to the screen. If the subroutine has been compiled with the N compiler option on, the disassembly will tell the subroutine name and the offset for each line of the disassembly. When you are done looking at your code, type G and hit Return. This will complete the execution of the **Step**. For more information on Macsbug see Appendix G.

Note: If Macsbug is not installed, a Command-Shift-S is identical to choosing **Step** from the **Run** Menu.

An Example

The following example shows how you can use LightsBug to debug a program in a manner that would be impossible with just the Instant and Observe Windows. Included on a Lightspeed Pascal disk is the sample program "Parser". Parser is a program which computes the value of Pascal integer expressions. But there is a bug in this program! We will show how LightsBug provides valuable assistance in debugging this more complex program.

Begin by creating a project by choosing **New Project...** from the **Project** Menu. Then choose **Add File...** from the **Project** Menu to add Parser to the project. Choose **Go** from the **Run** Menu to execute Parser. When the program prompts you for an expression, type in $2+5*(6+4*3)$ and hit the Return key. The program will calculate the value of the expression and display the result. It will look similar to this:



The calculated result of 60 is not the correct value for the expression (92 is the correct value). This means that there is something wrong with the program. In general, debugging is not an easy task. But when you have a program like Parser, which uses recursion to evaluate expressions, debugging is even more difficult.

LightsBug is designed to provide valuable assistance in this debugging task. To see how LightsBug helps, you should execute the program again. First stop the running program by clicking on the Bug Spray Can in the right end of the menu bar. Then choose **Reset** from the **Run** Menu. Open an editing window for the Parser by double-clicking on the filename in the project Window. Next choose **Stops In** from the **Debug** Menu and put a Stop Sign on the **case** statement in the procedure *Term*. And finally, run your program again, typing in the same input as before. When your program stops, bring up the LightsBug Window. Click on the second instance of the routine *Term* in the subroutine call chain pane. The display will look similar to this:

Parser

```

{Since the multiplying operators are left a
while Sy = MulOp do
begin
  SavedValue :=
  SavedOp := 0;
  InSymbol;

  Factor;

  {Now multiply
case SavedOp
  Mul :
    Value := S
  IntDiv :
    Value := S
  IntMod :
    Value := S
end;
end; {while}
end; {Term}

```

Text

Enter an expression:
2+5*(6+4*3)

LightsBug

registers	Term
zones	SavedOp 0100 0000
Term	SavedValue 0000 0005
Expression	Expression
Factor	MinusSigned 0004 0000
Term	SavedOp 0400 0000
Expression	SavedValue 0000 0000
ParsedOK	ParsedOK
Parser	function value 013A 0004
	ParseString 00FF 0004
	Global variables
	Ch 0000 0001
	InputString 0C32 2B35
	Offset 000D 0000
	Op 0001 0000
	ReservedWordTab... 0344 4956
	Sy 0100 0000
	Value 0000 0003

Base: 00000 + Offset: 0000 Edit

000:	0000	0000	0004	46B0	0002	1B54F	...T
00C:	0002	442C	0004	457C	0004	46B0	..D,..E ..F	
018:	0002	4514	0004	46B0	0100	0C32	..E...F	...2
024:	2B35	2A28	362B	342A	3329	0014	+5*(6+4*3)...	

At this point in the program, the input string has been scanned and the values for all five numbers in the expression (2,5,6,4, and 3) should be stored in variables. Careful examination of the program will reveal that the 2 should be stored in the variable *SavedValue* of the first activation of the routine *Expression*, and the 5 should be stored in the variable *SavedValue* of the first activation of the routine *Term*. The 6 should be stored in the variable *SavedValue* of the second activation of the routine *Expression*, and the 4 should be stored in the variable *SavedValue* of the second activation of the routine *Term*. And finally, the 3 should be in the global variable *Value*.

When you look at the first activation of *Term* with LightsBug, you find that while the variable *SavedValue* of *Term* has the correct value of 5, the variable *SavedValue* of *Expression* has the value 0, not the expected value of 2.

Now click on the first instance of the routine *Term* in the subroutine call chain pane. The display will look similar to this:

Parser

```

{Since the multiplying operators are left a
while Sy = MulOp do
begin
  SavedValue :=
  SavedOp := Op
  InSymbol;

  Factor;

  {Now multiply
case SavedOp
Mul :
  Value := S
IntDiv :
  Value := S
IntMod :
  Value := S
end;
end; {while}
end; {Term}

```

Text

Enter an expression:
2+5*(6+4*3)

LightsBug

registers	Term
zones	SavedOp 0100 0000
Term	SavedValue 0000 0004
Expression	Expression
Factor	MinusSigned 0004 0000
Term	SavedOp 0400 0000
Expression	SavedValue 0000 0000
ParsedOK	ParsedOK
Parser	function value 013A 0004
	ParseString 00FF 0004
	Global variables
	Ch 0000 0001
	InputString 0C32 2B35
	Offset 000D 0000
	Op 0001 0000
	ReservedWordTab... 0344 4956
	Sy 0100 0000
	Value 0000 0003

Base: 04457E
Offset: 0000
Edit

```

000: 0000 0000 0004 46B0 0002 1B54 .....F...T
00C: 0002 442C 0004 457C 0004 46B0 ..D...E|...F
018: 0002 4514 0004 46B0 0100 0C32 ..E...F...2
024: 2B35 2A28 362B 342A 3329 0014 +5*(6+4*3)..

```

Again, while the variable *SavedValue* of *Term* has the correct value of 4, the variable *SavedValue* of *Expression* has the value 0, not the expected value of 6. This may lead you to suspect that the variable *SavedValue* of *Expression* is not getting a value assigned to it. In fact, this is the case. In this example, the line assigning a value to *SavedValue* has been intentionally commented out.

Find this line in the routine *Expression*, and remove the comment braces from around the assignment statement. Now run the program again (don't save the changes to your program, unless you want to permanently fix the bug). When the program hits the Stop Sign, the LightsBug display of the first activation of *Term* will look like this:

Parser

{Since the multiplying operators are left a

while Sy = MulOp **do**

begin

 SavedValue :=

 SavedOp := Op

 InSymbol;

 Factor;

 {Now multiply

case SavedOp

 Mul :

 Value := S

 IntDiv :

 Value := S

 IntMod :

 Value := S

end;

end; {while}

end; {Term}

Text

Enter an expression:

2+5*(6+4*3)

LightsBug

registers	Term
zones	SavedOp 0100 0000
Term	SavedValue 0000 0005
Expression	Expression
Factor	MinusSigned 0004 0000
Term	SavedOp 0400 0000
Expression	SavedValue 0000 0002
ParsedOK	ParsedOK
Parser	function value 013A 0004
	ParseString 00FF 0004
	Global variables
	Ch 0000 0001
	InputString 0C32 2B35
	Offset 000D 0000
	Op 0001 0000
	ReservedWordTab... 0344 4956
	Sy 0100 0000
	Value 0000 0003

Base: 00000 + Offset: 0000 Edit

000:	0000	0002	0004	46B0	0002	1B54F...	T
00C:	0002	442C	0004	457C	0004	46B0	..D,..E ..F	...
018:	0002	4514	0004	46B0	0100	0C32	..E...F...	2
024:	2B35	2A28	362B	342A	3329	0014	+5*(6+4*3)...	

The variable *SavedValue* of the first activation of *Expression* now has the correct value of 2.

Now select the second activation of *Term*. The display will look like this:

Parser

{Since the multiplying operators are left associative
while Sy = MulOp **do**
 begin
 SavedValue :=
 SavedOp := Op
 InSymbol;
 Factor;
 {Now multiply
 case SavedOp
 Mul :
 Value := S
 IntDiv :
 Value := S
 IntMod :
 Value := S
 end;
 end; {while}
end; {Term}

Text

Enter an expression:
 2+5*(6+4*3)

LightsBug

registers	Term	
zones	SavedOp	0100 0000
Term	SavedValue	0000 0004
Expression	Expression	
Factor	MinusSigned	0004 0000
Term	SavedOp	0400 0000
Expression	SavedValue	0000 0006
ParsedOK	ParsedOK	
Parser	function value	013A 0004
	ParseString	00FF 0004
	Global variables	
	Ch	0000 0001
	InputString	0C32 2B35
	Offset	000D 0000
	Op	0001 0000
	ReservedWordTab...	0344 4956
	Sy	0100 0000
	Value	0000 0003

Base: 00001E + Offset: 0000 Edit

000:	0000	0006	0004	454E	0002	1B54EN...	T
00C:	0002	442C	0004	451C	0004	454E	..D,..E...	EN
018:	0002	41AA	0004	46B0	0004	4570	..A™..F™..Ep	
024:	0002	1DA2	0002	419A	0004	453E	...¢..Aö..E>	

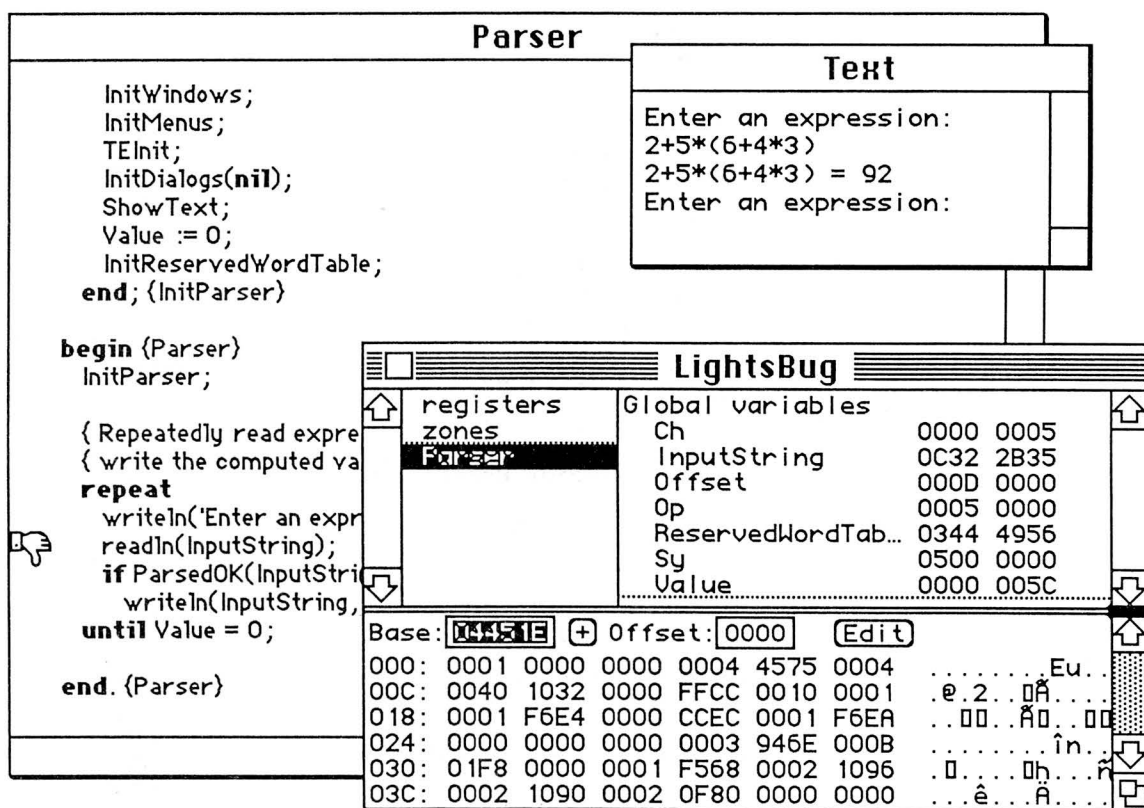
The variable *SavedValue* of the second activation of *Expression* now has the correct value of 6.

Now remove the Stop Sign from the program and continue execution. The display will look like this:

User's Guide

12-15

LightsBug



The program now calculates the correct result of 92.

In summary, this example demonstrated a number of the powerful features of LightsBug:

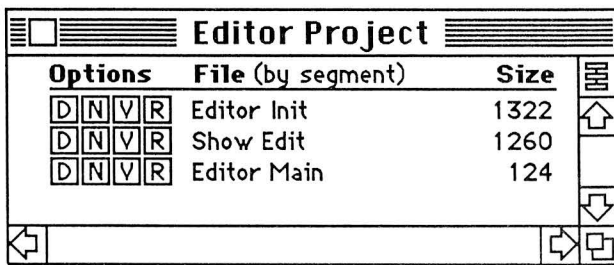
- 1) All program variables and their values are displayed in a easily readable manner.
- 2) The subroutine call chain allows you to see which routines are active and the order in which they were called.
- 3) Variables from different activations of recursive routines can be easily accessed.
- 4) Variables of the same name but in different routines can be displayed and easily differentiated.

Chapter 13

Compile Options

This chapter describes the various compile options that can be set or disabled from the project window, as well as related compile directives that you can embed directly into your program. So far, we've chiefly talked about the Debug option, which must be on for Lightspeed Pascal's debugging tools to work. This chapter goes into more detail on the other options and directives as well.

There are four compile options: D (Debug), N (Names), V (Overflow), and R (Range Checking). Whenever you add a file to the project, you will see four little boxes, labeled D, N, V, R, next to the name of the file. This display indicates that all four compile options are enabled by default.



Options	File (by segment)	Size
<input type="checkbox"/> D <input type="checkbox"/> N <input type="checkbox"/> V <input type="checkbox"/> R	Editor Init	1322
<input type="checkbox"/> D <input type="checkbox"/> N <input type="checkbox"/> V <input type="checkbox"/> R	Show Edit	1260
<input type="checkbox"/> D <input type="checkbox"/> N <input type="checkbox"/> V <input type="checkbox"/> R	Editor Main	124

To disable an option, simply click on the appropriate box. The box will disappear. To re-enable the option, click on the letter. The box will reappear.

In addition to the four compile options, Lightspeed Pascal includes a number of compile directives, which you can enable or disable from within your code. Compile directives are specified within comments, and should be placed on the line before the statement at which you want them to take effect.

Any comment whose first character is a \$ is assumed to be a directive. (The editor forces these directives to appear on a line by themselves.) The following directives are implemented:

<code>{ \$D+ }</code> or <code>{ \$D- }</code>	Turn Debug option on or off
<code>{ \$N+ }</code> or <code>{ \$N- }</code>	Turn Names option on or off
<code>{ \$R+ }</code> or <code>{ \$R- }</code>	Turn Range Checking option on or off
<code>{ \$V+ }</code> or <code>{ \$V- }</code>	Turn Overflow checking option on or off
<code>{ \$I+ }</code> or <code>{ \$I- }</code>	Turn automatic initialization on or off (only valid in main program)
<code>{ \$A+ }</code> or <code>{ \$A- }</code>	Turn special handling for asynchronous procedures on or off

Unknown directives are ignored.

As you can see, the first four directives correspond to the four compile options in the Project Window. The difference is that the directives allow you to affect code generation at the procedural level or even the statement level. For example, you can turn off checking in a time critical piece of code.

If a compile option is turned off for a file, then that option is turned off for that entire file. If a compile option is turned on for a file, then that option is initially turned on, but can be further turned on and off with the compiler directives.

The Debug, Name, or Asynch Directives must be turned on or off before the first **begin** of a procedure or function in order for it to take effect in that procedure or function. If you turn on/off Debug, Name, or Asynch in the middle of a subroutine, the directives affect the next subroutine, not the current one.

Note: The Lisa Pascal `{ $D± }` directive corresponds to the Lightspeed Pascal `{ $N± }` directive, not the Lightspeed Pascal `{ $D± }` directive.

How Compile Options Work

In general, compile options affect the way your code behaves during runtime. They do this by weaving "invisible" code into your code. For example, one option (D) allows you to use Lightspeed Pascal's powerful debugging features, while other options (V and R) insert error checking code for errors that can only be detected during runtime. The R option also detects some types of errors during compile time. If you turn on multiple options, then the code generated is a combination of the code generated by each option.

When you change a compile option for a file, the project will be rebuilt before you run.

D-Debug Option

The Debug option will:

- allow you to use all of Lightspeed Pascal's debugging features (see Chapter 7 and Chapter 12).
- display the line on which any runtime error occurs. Lightspeed Pascal will place a downturned thumb next to the statement where the error was detected. You then can observe or change the state of your program with the Observe and Instant windows. Examples of runtime errors are division by zero, reading beyond the end of file, and range errors (described below). If a runtime error occurs when the Debug option is off, then Lightspeed Pascal only shows the unit that the error occurred in.
- turn on the runtime stack checking for each subroutine compiled with this option. This guarantees that the stack hasn't collided with the application's heap (or anything else). If the stack collides with the heap, you may corrupt the Lightspeed Pascal environment, get a System Bomb, ID = 28, or witness hard-to-reproduce errors. This can save you from catastrophic errors.

In general, enabling the Debug option will greatly enhance your ability to find and fix problems in your programs.

Cost: Code size is increased by about 30%. Execution is about 50% slower.

Limits: This option can't be used in a standalone application or library. When you do a **Build & Save As...** application or library, all files with the Debug option turned on are automatically recompiled with the Debug option turned off.

(If you plan to use Macsbug to debug a piece of code, you will find that it is much easier to view 68000 instructions with the IL command (Instruction List) if the code has been compiled with the Debug option turned off.)

N-Name Option

The Name option embeds the first eight characters of each subroutine name into the code following the *RTS* instruction at the end of the code for the function or procedure. These names are visible to Macsbug and LightsBug. (See Chapter 12 for information on LightsBug or Appendix G for information on Macsbug.) If you do not intend to use Macsbug or LightsBug, then there is no need to use this option.

Cost: About 8 extra bytes in size per function or procedure. There is no effect on speed.

Limits: This option is only useful in conjunction with Macsbug or LightsBug. Names are uppercased and truncated to 8 characters; underscores are removed. If a name is shorter than 8 bytes, the data where the Macsbug name is stored is padded with spaces.

V-Overflow Option

The Overflow option generates code that detects errors resulting from numeric overflow in arithmetic operations. The result of each intermediate operation must be within the bounds of numerical representation. For type *integer*, these bounds are -32768 to +32767. For type *longint*, this range is -2147483648 to +2147483647. The Overflow option does not check floating point operations; however the SANE library does. The SANE environment allows you to control the effects of overflow in floating point operations. See Appendix D for more details.

For example, if you take two numbers 20000 and 21000 of type *integer* and add them together, you might expect to get 41000. However, because the largest number that can be represent by the type *integer* is 32767, the result will overflow (-24536) and will be wrong. If you have the Overflow option enabled, this error will be caught.

Cost: All arithmetic operations are slower. Size of generated code is increased.

Limits: Error recovery is awkward. Once an error is detected during runtime, a TRAPV instruction is executed, and execution is halted. When running under Lightspeed Pascal with the Debug option, you can observe and change values that cause the Overflow Error, and then continue. When you are running your

program as a standalone application, you can examine and change values with Macsbug, and then continue (but with more difficulty). However, you shouldn't expect the users of your program to do so. In situations that may lead to Overflow Errors you should explicitly check and handle the errors appropriately.

Examples of Overflow checking are shown below in three columns. The first column shows the original Pascal code that you would type in. The second column shows the rough, equivalent Pascal code that you would have to type in order to get the same result. The third column shows the actual 68000 assembly code generated by each option.

The equivalent Pascal code shouldn't be taken literally because it can't match the tricks done in assembly language. It is there to indicate the sort of thing being done by the 68000 code.

Original	Equivalent Pascal	68000
procedure P1; var i,j:integer begin	procedure P1; var i,j:integer; begin	LINK A6,\$FFFF0 ...
 i:= i+3;	 if i+3 > 32767 then TRAPV; i:= i+3;	 MOVE.W \$FFEE(A6),D0 ADDQ.W #3,D0 TRAPV ;check overflow MOVE.W D0,\$FFEE(A6)
 j:=j-4;	 if j-4 < -32768 then TRAPV; j:=j-4;	 MOVE.W \$FFEC(A6),D0 SUBQ.W #4,D0 TRAPV ;check overflow MOVE.W D0,\$FFEC(A6)
 i:=j*i+6;	 if j*i > 32767 then {multiply overflow} TRAPV; if (j*i)+6 > 32767 then TRAPV; {add overflow} i:=j*i+6;	 MOVE.W \$FFEE(A6),D0 MULS.W \$FFEC(A6),D0 DIVS.W #1,D0 TRAPV ;check overflow ADDQ.W #6,D0 TRAPV ;check overflow MOVE.W D0,\$FFEE(A6)
end;	end;	UNLK RTS

R-Range Option

The Range option does several kinds of checking. It makes sure that:

- a) The value being assigned to a variable is within the bounds of the variable's type. For the type *integer*, the range is $\pm\text{Maxint}$ (± 32767). (Due to the way Pascal defines type *integer*, -32768 is NOT a legal value.) For the type *longint*, the range is $\pm\text{MaxLongint}$ (± 2147483647).
- b) The index being used in an array access is within the bounds of the index type. When the Range option is on and the index is a constant, the range is checked during compile time. In the special case of accessing an element of a string, a check is done to see if the character being accessed is within the range of the string's current length ($\text{length}(\text{MyString})$), not its size ($\text{sizeof}(\text{MyString}) - 1$). For example, if `s25:string[25]` and `s25:='abc'`, then `s25[4]:='x'` will cause a range error. See Section 4.3.1 of the *Language Reference*.
- c) Before a pointer is dereferenced, the pointer value is not *nil*. This option does not do any further checks on pointer validity. In particular, potential odd address errors are NOT checked.

Cost: All operations that involve memory access, as described above, are slower and take up more space.

Limits: There is no graceful error recovery. Once an error is detected during runtime, a TRAP or TRAPV instruction is executed, and execution is halted. When running under Lightspeed Pascal with the Debug option, you can observe and change values that cause the Range Error, and then continue. When you are running your program as a stand-alone application, you can examine and change values with Macsbug, and then continue (but with more difficulty). However, you shouldn't expect the users of your program to do so. In situations that may lead to Range Errors you should explicitly check and handle the errors appropriately.

The example on the following pages illustrates the effective code generated for six statements with Range checking.

```

type
  PtrToInt = ^integer;
  ArrayType = array[3..5] of integer;
procedure P(x:integer;
            s:string;
            pi: PtrToInt);
var
  i:-6..7;      { size of i is one byte }
  a:ArrayType;
  s25:string[25];
begin

```

	if (x< -6) or (x>7) then	MOVE.W \$10(A6),D0
	{assignment out of range}	ADDQ.W #6,D0
	CHK;	CHK.W #13,D0
i:=x;	i:=x;	MOVE.B \$11(A6),\$FEEF(A6)

	if (x< 3) or (x>5) then	MOVE.W \$10(A6),D0
	{index out of range}	SUBQ.W #3,D0
	CHK;	CHK.W #2,D0
a[x]:=7;	a[x]:=7;	(assign 7 to a[x])

	if (x< 3) or (x>5) then	MOVE.W \$10(A6),D0
	{index out of range}	SUBQ.W #3,D0
	CHK;	CHK.W #2,D0
	if (a[x]<-6) or (a[x]>7) then	(move a[x] into D1)
	{assignment out of range}	ADDQ.W #6,D1
	CHK;	CHK.W #13,D1
i:=a[x];	i:=a[x];	(move a[x] into i)

	if length(s)>25 then	PEA \$FEF0(A6)
	{s too long to for s25}	PEA \$FECE(A6)
		JSR CheckandCopy
		BCC.S @1
	TRAP2;	TRAP #2 ;Range Error
s25:=s;	s25:=s;	@1:

	if (length(s25)<x) or (x < 1) then {string index out of range}	MOVE.W \$10(A6),D0 PEA \$FECE(A6) MOVE.W D0,-(A7) JSR CheckStrLength BCC.S @2
s25[x]:='c';	TRAP2; s25[x]:='c';	@2: TRAP #2 ;Range Error (move 'c' into s25[x])

	if pi = nil then {nil dereference}	MOVEA.L \$8(A6),A0 MOVE.L A0,D0 BNE.S @3
	TRAP0; if x < -32767 then	@3: TRAP #0 MOVE.W \$10(A6),D0 SUBQ.W #1,D0
pi^:=x;	TRAPV; pi^:=x;	TRAPV; Range Error MOVE.W \$10(A6),(A0)

end;	end;	UNLK RTS
-------------	-------------	-------------

The Range option also detects some compile time errors. For example, the following code will not compile with the Range option turned on, because it accesses the zeroth element of a string (which is the string's length byte). It will compile with the Range option turned off.

```
{function MyLength is equivalent to predefined function Length}

function MyLength (s:string): integer;
begin
  MyLength:= ord(s[0]); {Compiles ONLY if Range option is off}
end;
```

The Initialization Directive

As discussed in Chapter 9, {\$I+} and {\$I-} can be used to enable or disable the automatic Toolbox initialization calls inserted by Lightspeed Pascal. You should only need to turn this feature off if you have already explicitly inserted the initialization calls into your program, and even then, only when you are saving your project as an application.

If you use this directive, it must appear before the **begin** of your main program. This directive is ignored in a unit.

The Async Directive

The Async (A) compiler directives `{ $A+ }` and `{ $A- }` are used when compiling asynchronous procedures.

`{ $A+ }` or `{ $A- }` Turn Async register saving On/Off

A procedure compiled with this directive on will save A5 in addition to other saved registers. Also, register A5 will be copied from the low memory global CurrentA5.

You can achieve the same effect using the SetupA5 and RestoreA5 built-ins. The `{ $A }` directive is simply a more elegant automatic form of Setup/RestoreA5.

The Async directive automatically turns off the Debug (D), Range (R) and Overflow (V) compile options.

Using Directives

The Debug, Name, or Async Directives must be turned on or off before the first ***begin*** of the procedure or function in order for it to take effect in that procedure or function. If you turn on/off the Debug, Name, or Async directives in the middle of a subroutine, the directives affect subsequent subroutines, not the current one.

Example:

```
procedure Async_Proc_Without_Debug;

{declarations}

{ $A+ }
{ $N- }

begin
{ The next two directives do not affect this procedure }
{ but will affect all subroutines following. }
{ $A- }
{ $N+ }

{rest of procedure}

end;
```

Chapter 14

Inside Lightspeed Pascal

This chapter gives a look inside Lightspeed Pascal, and explains a little bit about how it works its magic. You need not read this chapter to use Lightspeed Pascal; it is provided for your enjoyment and deeper understanding.

Two Worlds

Lightspeed Pascal supports two virtually separate Macintosh environments. The first environment is the edit-compile-build world with which the user interacts while preparing a project. The second environment is the user world, within which the user's project executes. The objective is to create, as much as possible, the illusion of a stand-alone Macintosh, yet still support the full repertoire of Lightspeed Pascal debugging features.

There are four important events during the execution of a user's project.

- 1) Creation and initialization of the user world.
- 2) Clean-up and disposal of the user world.
- 3) Context switch TO the user world FROM the Lightspeed Pascal environment.
- 4) Context switch FROM the user world TO the Lightspeed Pascal environment. The last two of these may occur any number of times for a single invocation of the project.

Creation and Initialization of the User World

The user world is created and initialized immediately prior to executing the project. The user world consists of a pointer block in Lightspeed Pascal's application zone, a handled buffer containing various low-memory globals which are not shared between the two worlds, and reserved memory for the user world's registers.

The size of the pointer block is equal to the sum of the project's zone size and stack size (specified via **Run Options...**), plus the size required for the user's A5-world (jump table and globals). The user's application zone occupies the lowest memory of the pointer block, followed by the user's stack, followed by the user's A5-world. The A5-world is initialized with the jump table and a copy of the current Lightspeed Pascal QuickDraw globals, which are prepared with the Drawing Window port as the current port, and the cross-hair cursor as the current cursor.

The size of the handled buffer is determined by the amount of memory which is not shared between the two worlds, as specified by the Lightspeed Pascal resource *QPRF 2* (*QPRF 3* if *ROM85* is true). This resource is an array of pairs of integers, each pair representing the starting address of a block to buffer and the size of that block. The addresses must be in increasing order and the blocks may not overlap.

The creation of the user's low-memory buffer is a four-step process:

1. Copy specified low-memory locations into Lightspeed Pascal's buffer.
2. Set low memory to initial values appropriate for the user's world, as follows:
 - a. Set *TopMapHndl* to the user's project, and open the resource file specified by the Run-Time Environment Settings, if any.
 - b. Clear the menu bar.
 - c. Clear the four handles to *DAStrings*.
 - d. Clear *IAZNotify*.
 - e. Set *ApplZone*, *ApplLimit* and *HeapEnd* as appropriate for the user's zone.
3. Copy specified low-memory locations into the user's buffer.
4. Restore specified low-memory locations from Lightspeed Pascal's buffer.

Finally, the reserved memory representing the user's registers is initialized, as follows: *A5* is set to the base of user's *A5*-world. *A6* is set to the initial *A5*. *A7* is set to the top of the user's stack, below the initial *A5* by the amount required for user globals. *PC* is set to *_MAIN*, in the jump table.

Clean-up and Disposal of the User World

Disposal of the user world may occur due to a Reset or to execution of an *ExitToShell* from within the user program. The clean-up occurs in distinct steps, in this order:

1. Close windows. For all windows in the user-world pointer block, either *CloseWindow* or *CloseDeskAcc*, as appropriate.
2. Context switch TO the user-world, without enqueueing VBL tasks.
3. Close resource files and clean up resource maps.
 - a. *CloseResFile* for any resource file whose map is in the user-world pointer block,
 - b. *ReleaseResource* for any resource whose master pointer is in the user-worldpointer block,
 - c. *EmptyHandle* for any resource whose data resides in the user-world pointer block.

4. Execute `_TERM` from the appropriate system library (e.g. *MacPasLib*). This may, for example, close files opened via Pascal I/O.
5. Execute the *IAZNotify* routine, if present. Recall that *IAZNotify* was zeroed in the user world at initialization. Note that the *IAZNotify* routine is invoked after resource files have been closed. If the *IAZNotify* routine is in a CODE resource, that CODE resource should have been detached.
6. Release the memory occupied by the user-world pointer block. Upon completion, the project returns to the Reset state. The next request to execute will invoke the creation and initialization of the user world.

Context Switch TO the User World

Here's what happens during a switch to the user world from the Lightspeed Pascal environment.

First, the user's low-memory buffer is copied into Macintosh low-memory. Second, the menu bar is redrawn. (This may cause any menu bar changes instituted by the user program to appear BEFORE the user program has executed a *DrawMenuBar*.) Then, VBL tasks which had been dequeued during the context switch FROM the user world are enqueued.

Finally, the user's registers are restored from their saved values, the user PC is pushed, and an RTS is executed. A debugger trap (*\$ABFF*) will be executed right before this RTS if the context switch was initiated by Command-Shift-S, and if a machine-level debugger has been installed.

Context Switch FROM the User World

Here's what happens during a switch from the user world to the Lightspeed Pascal environment.

First, VBL tasks whose code resides in the user-world are dequeued. Second, the Macintosh low-memory is copied to the user's low-memory buffer, and low-memory is restored from the Lightspeed Pascal low-memory buffer. Then, the values of the registers are copied into reserved memory, to be restored at the next context switch TO the user world. The register values may be examined using LightsBug's register display.

Note: The values saved for the PC and A7 are those values recorded at the last statement executed which had been compiled with the Debug option on.

Then, if no error has been detected, extra checking is performed. This checking includes quick heap consistency checks on both world's application zones and on the system zone. If an error is detected during these checks, the Finger will be changed to a thumb and an error message will be presented (if possible!).

Finally, the Finger (or thumb and error message) is displayed in the appropriate window, indicating the latest statement executed which had been compiled with the Debug option on. If it is NOT an A-Trap break (see below for A-Trap breaking), the next context switch TO the user world will begin execution at the statement indicated by the Finger.

Warning: If the context switch is forced by an exception which occurs in code that was compiled with the **Debug** option off, many statements which have already been executed will be re-executed when the next **Run** command is issued.

Context Switches for A-Trap Breaking

Context switches for the 'Break at A-Traps' option of the **Debug** Menu are different with respect to saving and restoring registers. The statement indicated by the Finger will still be the latest statement executed which had been compiled with the Debug option on. However, the register values saved are the values AT THE POINT of the A-Trap. Execution will continue at the A-Trap itself, and NOT at the statement indicated by the Finger.

Context Switching and Completion Routines

No special handling is performed to accomodate completion routines through context switching or user-world disposal. This implies that, if the user program installs a completion routine, it might not be executed in the user world. Thus A-Trap AND exception behaviors may be different in a completion routine.

The completion routine must have been compiled with the Debug option off, and should not depend on the values of any of the swapped low-memory globals (including *CurrentA5*). It is recommended that the user program use the 12-byte area *ApplScratch* for communication with such routines. Lightspeed Pascal neither modifies *ApplScratch*, nor swaps it during a context switch. It is also recommended that the user program install an *IAZNotify* routine to clean up its own mess before the user-world is reclaimed for use by Lightspeed Pascal .

Bottlenecks

Several A-Trap handlers are customized by Lightspeed Pascal, for either of two reasons. *SetHandleSize*, *GetHandleSize* and *PostEvent* are modified just to improve the performance of Lightspeed Pascal . The other bottlenecks are installed to facilitate running and debugging in the dual environment. The special handlers are installed during application initialization and removed via the *IAZNotify* mechanism during the launch of the next application.

Several of the modified A-Traps may cause your program to execute somewhat differently within the Lightspeed Pascal environment than it would execute as a stand-alone application. The behaviors of *FrontWindow*, *FindWindow*, *GetNextEvent* and *EventAvail* may be especially surprising.

The remainder of this section is a description of the modified A-Traps. Note that this section assumes familiarity with the unmodified behaviors. See *Inside Macintosh* for more details.

<i>RecoverHandle</i>	Since there are two application zones when the user program is running, it is not always correct to add the current zone to the base address to recover the handle. Lightspeed Pascal tries the current zone first, then the Lightspeed Pascal application zone. If neither of these appears to work, the last try is the user's application zone. Note that no error detection is performed by <i>RecoverHandle</i> .
<i>SetApplLimit</i>	NO OP.
<i>InitGraf</i>	Sets <i>randSeed</i> to 1. Note that if <i>InitGraf</i> is NOT executed, <i>thePort</i> corresponds to the Drawing Window, and <i>randSeed</i> is unchanged since the last time a user program was executed.
<i>InitFonts</i>	NO OP.
<i>InitWindows</i>	NO OP.
<i>SelectWindow</i>	Disable the <i>FrontWindow</i> bottleneck, since the window manager uses <i>FrontWindow</i> to determine which window to deactivate.
<i>FrontWindow</i>	Scan the window list from front to back for a window whose storage is in the user's world. If none is found, return NIL. If one is found, and it is already front, simply return its pointer. If one is found and it is not front, bring the window to front and return its pointer. If deactivation of a Lightspeed Pascal Window is consequentially required perform the deactivation before returning.
<i>FindWindow</i>	Execute the ROM's <i>FindWindow</i> , and, if the window found belongs to Lightspeed Pascal, perform the appropriate select or drag, set the window to NIL and return the code <i>inDesk</i> .
<i>InitMenus</i>	NO OP.
<i>DrawMenuBar</i>	If executing the user's program, draw the Bug Spray Can after drawing the menu bar.
<i>GetNextEvent</i>	Execute the ROM's <i>GetNextEvent</i> . If the returned event is an activate or update event for a Lightspeed Pascal Window, handle the event and try again.
<i>EventAvail</i>	Execute the ROM's <i>EventAvail</i> . If the returned event is an activate or update event for a Lightspeed Pascal Window, handle the event and try again.
<i>InitDialogs</i>	NO OP.

<i>SystemError</i>	If the error code is between -1024 and -4095, halt the user program and present an error message. Some of the error codes are reserved by Lightspeed Pascal, as follows: -2001..-2197 correspond to Macintosh system error codes -1..-197. -3001..-3100 and -4001..-4095 are run-time errors detected by Lightspeed Pascal. The rest of the error codes are left for the user program to define.
<i>TEInit</i>	NO OP.
<i>ExitToShell</i>	Perform a context switch and reset the user's world.

Exceptions

Many of the 68000 exceptions are intercepted by Lightspeed Pascal before proceeding with standard Macintosh exception handling. The exception vectors are installed during Lightspeed Pascal application initialization, and removed during application termination via the *IAZNotify* mechanism.

The following exceptions are intercepted by Lightspeed Pascal :

- Address Error
- Illegal Instruction
- Zero Divide
- CHK Instruction
- TRAPV Instruction
- Line 1010 Emulator
- Line 1111 Emulator
- Level 4 Interrupt Autovector †°
- Level 5 Interrupt Autovector †°
- Level 6 Interrupt Autovector †°
- Level 7 Interrupt Autovector †
- TRAP #\$0 Instruction
- TRAP #\$1 Instruction
- TRAP #\$2 Instruction
- TRAP #\$3 Instruction
- TRAP #\$4 Instruction
- TRAP #\$5 Instruction °
- TRAP #\$6 Instruction
- TRAP #\$7 Instruction
- TRAP #\$8 Instruction
- TRAP #\$9 Instruction
- TRAP #\$A Instruction
- TRAP #\$B Instruction
- TRAP #\$C Instruction
- TRAP #\$D Instruction
- TRAP #\$E Instruction
- TRAP #\$F Instruction †

- † Not intercepted if a machine-level debugger has been installed. The machine-level debugger is detected by comparing the value of the Level 7 Interrupt Autovector to the low-memory global *ROMBase*. It is assumed that a machine-level debugger has been installed if *ROMBase* is the GREATER.
- Not intercepted if the host machine is a Macintosh XL.

If an exception occurs when the user program is running, the program is halted and an error message is generated. In some cases, the user can repair the cause of the exception using Instant or LightsBug, and proceed with program execution.

The Zero Divide, CHK Instruction (subrange error), TRAPV Instruction (integer overflow), Trap #0 (Nil dereference), TRAP #1 (subrange error), TRAP #2 (string range error), Trap #3 (set value range error), Trap #4 (case selector range error) and Trap #6 (for loop control variable modified) exceptions may all occur due to errors detected during program execution. The CHK, TRAPV and TRAP instructions are all emitted by the code generator for error detection. The Address Error exception may occur due to an undetected error during program execution, e.g. dereferencing an uninitialized pointer.

The Illegal Instruction, Line 1111 Emulator, TRAP #5, and TRAP #F exceptions all imply serious trouble: either some executable code has been overwritten, or an attempt has been made to execute data.

The Line 1010 Emulator exception intercept takes affect only when the user program is running and the **Break at A-Traps** item of the **Debug** Menu is checked. Lightspeed Pascal determines the location of the A-Trap (not in ROM, for example), and, if appropriate, forces a context switch after saving the machine state in a reserved area of memory. This Lightspeed Pascal A-Trap break will occur before the any Macsbug A-Trap break which may have been set.

Chapter 15

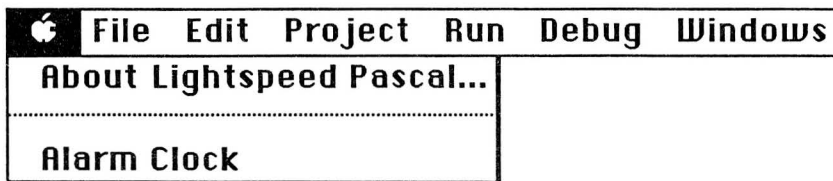
Menu Reference

Introduction

This chapter is a brief summary of the Lightspeed Pascal Menus.

All of Lightspeed Pascal's commands are briefly covered here. Additional coverage of more complex commands is provided elsewhere in this manual.

The Menu



About Lightspeed Pascal...

This command tells you specific information about the version of Lightspeed Pascal you're working with, as well as statistics about memory usage.

The other items in this menu are desk accessories which come with your Macintosh. See *Macintosh*, your owner's manual, for details.

The File Menu

🍏	File	Edit	Project	Run	Debug	Windows
	New		⌘N			
	Open...		⌘O			
	Close					
	Save					
	Save As...					
	Save a Copy As...					
	Revert					
	Page Setup...					
	Print...					
	Delete...					
	Transfer...					
	Quit		⌘Q			

New (⌘ N)

This command creates a new Pascal editing window with the name *Untitled*. You can save the contents of the window to a file using the **Save As...** menu command.

Open... (⌘ O)

This command allows you to open an existing Pascal file for editing. A maximum of eight Pascal editing windows may be open at once. It also allows you to open previously saved Instant and Observe Windows.

Close

This command allows you to close the active window. If you try to close an editing window, and the file has been modified since it was last saved, a dialog box will ask you if you want to save the changes, discard them, or cancel the **Close** command. Note that this command cannot be used to close the Project Window. Use the **Close** command on the **Project** Menu for that purpose.

Save

This command saves the file in the active edit window to disk. If the file is currently untitled, a dialog box will ask you to name the file.

Save As...

This command allows you to save the current file under another name. If you have made edits in the current session, they will be saved under the new name. The original file will remain unchanged, and as you continue editing, you will be editing the new file. This feature is useful for switching to a new version of a file, leaving the old file as a backup.

Note that when you use **Save As...**, the newly saved file replaces the old one in the project. Lightspeed Pascal preserves the tie between the file you are editing and the project. If the file appears in the Project Window, and if the new name doesn't already appear in the Project Window, then the entry for the file in the Project Window is changed to match the new file name. Use **Save a Copy As...** if you don't want this to happen.

Save As... is also used to store **Instant** and **Observe** Windows as files. When either of those windows is the active window, special dialog boxes appear for saving those windows.

Save a Copy As...

Unlike **Save As...**, this command does not affect the status of the file currently being edited; it simply "snapshots" it to another file. This is a good way to make backups without finding yourself editing the backup.

Neither **Save As...** nor **Save a Copy As...** will let you save over a file already open in an edit window or an existing Lightspeed Pascal Project or Library.

Revert

This command restores the last-saved version of the current file, discarding any edits made since the last **Save** or **Save As...**

Page Setup...

This command allows you to specify the size of the paper you're printing on, and whether the file should be printed upright on the page (tall orientation) or sideways (wide orientation).

If there is no printer resource in the system folder, a warning appears when you choose this command. See *Macintosh*, your owner's manual, for details.

Print...

This command allows you to print the file in the active window. A dialog box will prompt you for various settings.

Click **Cancel** to cancel the **Print...** command, or hold down the command key ⌘ while pressing the period (.) to stop printing that is in progress.

If there is no printer resource in the system folder, a warning appears when you choose this command. See *Macintosh*, your owner's manual, for details.

Delete...

This command allows you to delete a file from the file system without having to leave Lightspeed Pascal. You will be asked to confirm the deletion.

Transfer...

This command allows you to launch another application without first returning to the Finder or MiniFinder.

Quit (⌘ Q)

This command quits Lightspeed Pascal.

If you choose **Quit** while a project is open, it will automatically be closed. If any editing windows have been modified, it will ask if you want to save or discard your edits, or cancel the **Quit** command.

The Edit Menu

🍏	File	Edit	Project	Run	Debug	Windows
		Undo		⌘Z		
		Cut		⌘H		
		Copy		⌘C		
		Paste		⌘V		
		Clear				
		Select All				
		Show Selection				
		Show Error				
		Find		⌘F		
		Replace		⌘R		
		Everywhere		⌘E		
		Find What...		⌘W		

Undo (⌘ Z)

This Menu item is supplied only for compatibility with desk accessories which require it.

Cut (⌘ X)

This command removes selected text and places it in the Clipboard. It replaces the current contents of the Clipboard (if there are any).

Copy (⌘ C)

This command copies the selected text and places it in the Clipboard. The copy can now be pasted somewhere else using the **Paste** command (⌘ V).

Paste (⌘ V)

This command copies the contents of the Clipboard into the file being edited at the insertion point. If text is currently selected, it is replaced.

Clear

This command clears the selected text. The selection is not placed on the Clipboard, and cannot be recovered. You can also clear selected text by simply hitting the backspace key.

Select All

This command selects all the text currently in the window.

Show Selection

It is possible to scroll the active window so that the cursor or the selected text no longer appears on the screen. This command allows you to quickly return to the point in the file where the cursor or the selected region of text is positioned.

Show Error

When Lightspeed Pascal finds an error in your Pascal program, a downturned thumb appears in the editing window at the point in the file where Lightspeed Pascal believes the error to be. If you scroll such that the thumb no longer appears in the window, this command lets you return quickly to the point where the thumb appears.

Find (⌘ F)

This command searches for the string specified with the **Find What...** Menu item. If the string is found, it is selected, and the window scrolls to make the selection visible. Searching is done from the current insertion point to the end of the file.

Replace (⌘ R)

This command replaces the selected text with the replacement string specified with the **Find What...** Menu item.. If no replacement string has been entered in the dialog box resulting from the **Find What...** command, replace is inactive.

Everywhere (⌘ E)

This command replaces every instance of the search string with the replace string specified with the **Find What...** command. A dialog box will ask you to confirm that you really want to do this.

Find What... (⌘ W)

This command allows you to specify what characters are to be searched for when executing a **Find** or an **Everywhere**, and what characters are to be used for replacement when executing a **Replace** or an **Everywhere**. The dialog box for this command also lets you specify whether the search should be case sensitive, and whether to search for the selected characters as a discrete word, or search for any occurrence of the character string.

The Project Menu



New Project...

This command creates a new project document. An empty Project Window will appear. You can then add files to the project with the **Add File...** or **Add Window** commands. Only one project can be open at a time.

Open Project...

This command is used to open an existing project.

Close Project

This command closes the currently open project.

If you try to close a project with opened files whose latest versions have not been saved, you will be asked if you want to save your edits.

Add Window

This command adds the active Pascal editing window to the project. If an untitled window is added, a dialog box will appear, asking you to save the contents of the window in a file.

Add File...

This command lets you select Pascal files to add to the project. Four kinds of files will be accepted:

- 1) *Text* files.
- 2) Files created with the Lightspeed Pascal editor.
- 3) Libraries created by Lightspeed Pascal.
- 4) Libraries created by other development systems that have been processed by the .Rel Converter utility.

Remove

This command lets you remove the currently selected source file or library from the project.

Build & Save As...

This command compiles your project and saves it in one of three forms:

- 1) As a double-clickable application. The Debug compile option is automatically disabled.
- 2) As a library. The Debug compile option is automatically disabled.
- 3) As a *compressed* project. With this option, the information in the project is saved without the object code for the project. This is handy for making backups.

View Options...

This command allows you to customize the layout of the Project Window. There is also a brief description of Lightspeed Pascal's compilation options.

Run Options...

This command allows you to perform three distinct tasks:

- 1) Select a Macintosh Resource file to be used by your program. See Chapter 10, *Using Resources*, for more information.

- 2) Adjust parameters of the window used for output text from your program: how many characters of the text window are saved (and are viewable by scrolling the text window), and whether to echo text window output to a printer or file. You can also choose the text window font. See Chapter 6, *Running a Program*, for more information.
- 3) Adjust the application stack and zone sizes within Macintosh memory. See Chapter 6, *Running a Program*, for more information.

Source Options...

This command allows you to adjust how text will appear in the edit windows. You can specify the font, indentation width, and tab stop width in the editing windows. This information is saved with the project.

The Run Menu

🍏	File	Edit	Project	Run	Debug	Windows
				Check	⌘K	
				Build	⌘B	
				Check Link		
				Reset		
				Go	⌘G	
				Go-Go		
				Step	⌘S	
				Trace	⌘T	
				Auto-Save		
				✓Confirm Saves		
				Don't Save		

Check (⌘ K)

This command checks the Pascal syntax of the active editing window. It can also be used to check the syntax of the Instant or Observe windows.

Build (⌘ B)

This command in the **Run** Menu compiles all files that are currently tagged for re-compilation. **Go**, **Go-Go**, **Step**, or **Trace** perform an implicit build before executing the program.

Check Link

This command performs an explicit *link* on the project. Linking takes only a few seconds, and is usually performed automatically before you run. It is useful for finding undefined or multiply-defined symbols.

Reset

This command resets a paused program. The next **Step**, **Trace**, **Go** or **Go-Go** will start the program running from the beginning.

Go (⌘ G)

This command runs the program. If necessary, the project is built before running.

Go-Go

This command is used in conjunction with stops, which can be introduced into your program using **Stops In** from the **Debug Menu**. The program will run until it reaches a statement with a Stop Sign preceding it in the file window; it will then pause just long enough to update the values in the Observe Window, and continue executing.

Step (⌘ S)

This command runs the next statement in your program. The next statement is indicated by a Finger pointing to the statement (if the statement is in an open editing window).

Choosing **Step** with the Shift key pressed causes Lightspeed Pascal to break into Macsbug at an RTS instruction just before the actual code for the statement.

Trace (⌘ T)

This command can be thought of as an automatic version of **Step**. It executes your program at the rate of several statements a second: faster than you can specify with **Step**, but slow enough for you to be able to look and how the program's execution is proceeding.

Auto-Save

Confirm Saves

Don't Save

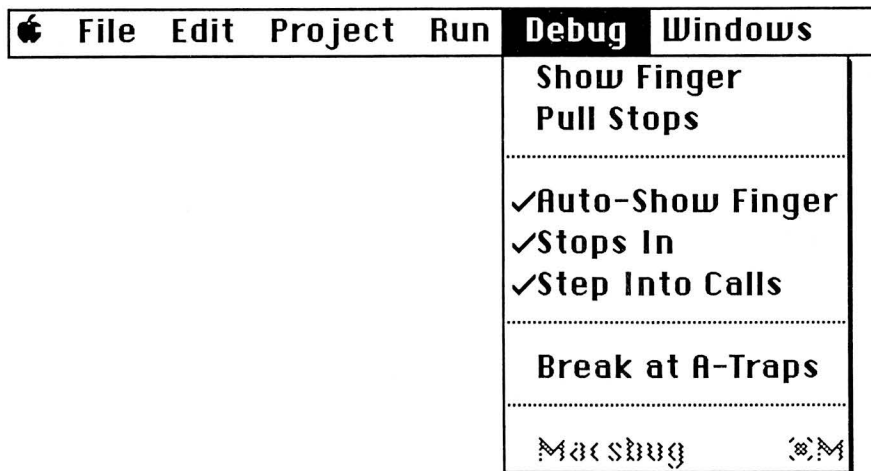
These three options are mutually exclusive; only one option is active at a time.

Choosing **Auto-Save** will automatically save all changed files prior to running your program. If you first hold down the shift key, and then choose **Auto-Save**, all your unsaved files will be automatically saved. The current option is not changed.

Confirm Saves asks if you want to save changed files prior to running your program. If you first hold down the shift key, and then choose **Confirm-Save**, Lightspeed Pascal will ask you if you want to save each of your unsaved files. The current option is not changed.

Don't Save specifies that you do not want to save files prior to running your program.

The Debug Menu



Show Finger

This command makes the window where the Execution Finger is located the active window, and, if necessary, scrolls the window contents to make the Finger visible. If the window where the execution Finger *would* be is not open, then the Project Window will become the active window, and the Execution Finger will point at the file name of the currently executing unit.

Pull Stops

This command removes all Stops from the currently active file window. Stops are placed in the active window with the **Stops In** command.

Auto-Show Finger

This command can be thought of as an automatic version of **Show Finger**. During a **Step**, **Go-Go**, or **Trace** execution, the window where the Execution Finger is located is always the active window. As execution passes from unit to unit, the editing window for the current unit is made the active window. If the unit where the Execution Finger *would* be is not opened, then the Project Window will become the active window, and the Execution Finger will point at the file name of the currently executing unit.

Stops In

This command allows you to introduce stops into your Pascal code. Used in conjunction with the **Go** or **Go-Go** command in the **Run** Menu, execution will stop at the statement which is preceded by the Stop Sign. the statement will not be executed until you continue running.

Once this option has been chosen, you can introduce Stop Signs into the active editing window by moving the pointer to the left hand border of the window, and clicking next to the statement where you want to place a Stop.

Step Into Calls

When **Step Into Calls** is on, the **Go-Go**, **Step**, or **Trace** run options will cause the Execution Finger to be shown as the program executes procedures or functions. When it is off, a procedure or function call is executed in its entirety without the Finger tracking the execution through the procedure.

Break at A-Traps

This command specifies that program execution should halt before execution of a Macintosh A-trap call. This is useful for checking that the proper values are being passed at the time of the call; you can then enter Macsbug to view the arguments to the A-trap in the processor registers or on the stack.

Macsbug (⌘ M)

This command enters the Macsbug debugger. It is disabled when Macsbug is not loaded.

The Windows Menu

🍏	File	Edit	Project	Run	Debug	Windows	
							Editor Project ⌘P
							LightsBug ⌘L
							Instant Observe
							Text Drawing
							Clipboard
							◊ Editor Main ⌘1
							◊ Untitled 1 ⌘2
							Available ⌘3
							Editor TopLevel ⌘4
							Available ⌘5

Editor Project (⌘ P)

This command makes the Project Window the active window.

LightsBug (⌘ L)

This command opens the LightsBug Debugging Window.

Instant

This command opens the Instant Window, which is used for executing statements when your program is paused.

Observe

This command opens the Observe Window, which displays the values of expressions during program execution.

Text

This command opens a window for text output from Write and Writeln statements.

Drawing

This command opens a window for drawing output from your program.

Clipboard

This command opens the Clipboard Window. The Clipboard is where selected text that has been **Cut** or **Copied** has been stored.

File Windows (⌘ 1 up to ⌘ 8)

Use these commands to activate one of the currently open editing windows. Note that in the menu, a diamond appears next to files which have unsaved edits.

PART TWO

LANGUAGE REFERENCE

Preface

This manual describes the programming language supported by Lightspeed Pascal. This language is intended to be compatible with Macintosh Pascal, Apple's LisaPascal, and American National Standard Pascal, inasmuch as it is possible to be compatible with all at the same time. Appendices A and B describe the differences between Lightspeed Pascal and these versions of Pascal.

Operating Environment

Lightspeed Pascal will operate on any Macintosh computer with at least 512K memory.

Related Documents

Pascal User Manual and Report, Third Edition (ISO Pascal Standard) by Jensen and Wirth, revised by Mickel and Miner, Springer-Verlag, New York, 1985.

American National Standard Pascal Computer Programming Language, ANSI/IEEE770X3.97-1983, IEEE/Wiley-Interscience, 1983.

Inside Macintosh, Addison Wesley, 1985.

Definitions

For the purposes of this manual, the following definitions are used:

<i>Error</i>	Any misuse of the Pascal language as described in this manual. <u>Most</u> such errors will be reported either before or during execution of a program. Other errors are not detected. The undetected errors are listed in Appendix B. The behavior of a program that contains an undetected error is unspecified (see below).
<i>Undefined</i>	A value of a variable or function that is not meaningful. It is an error to attempt to make use of an undefined value.
<i>Unspecified</i>	When a situation arises in the execution of a program where several courses of action are possible, often the specific course chosen is said to be unspecified. This means that the program should not depend on any specific course being chosen, as the result may be unpredictable. This leaves the implementation free to choose the course that is most convenient at the time.

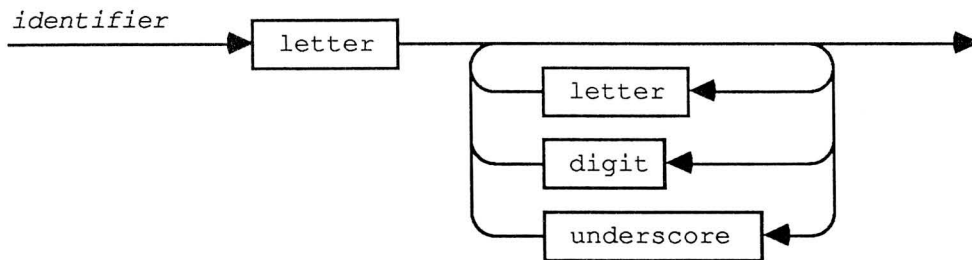
Notation and Syntax Diagrams

All numbers in this manual are in decimal notation, except where hexadecimal notation is specifically indicated.

Throughout this manual, *italics* are used to distinguish Pascal text from English text. Pascal word-symbols are displayed in bold-faced italics. For example, `sqr(n div 16)` represents a fragment of a Pascal program. Sometimes the same word appears both in plain text and in italics; for example, "The declaration of a Pascal procedure begins with the word ***procedure***."

Italics are also used when technical terms are introduced.

Pascal syntax is specified by diagrams. For example, the following diagram gives the syntax for an identifier:

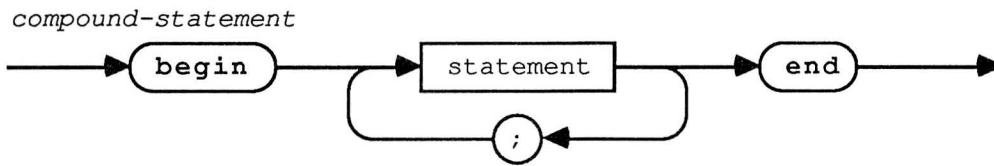


Start at the left and follow the arrows through the diagram. Numerous paths are possible. Every path that begins at the left and ends at the arrow-head on the right is valid, and represents a valid way to construct an identifier. The boxes traversed by a path through the diagram represent the elements that can be used to construct an identifier. Thus the diagram embodies the following rules:

- An identifier must begin with a *letter*, since the first arrow goes directly to a box containing the name "letter".
- An identifier might consist of nothing but a single letter, since there is a path from this box to the arrow-head on the right, without going through any more boxes.
- The initial letter may be followed by another *letter*, a *digit*, or an *underscore*, since there are branches from the initial letter box to these boxes.
- The initial letter may be followed by any number of letters, digits, or underscores, since there is a path that leads from these boxes back to them again.

A word contained in a rectangular box may be a name for an atomic element like "letter" or "digit," or it may be a name for some other syntactic construction that is specified by another diagram. The name in the rectangular box is to be replaced by an actual instance of the atom or construction that it represents, e.g. 3 for "digit" or *counter* for "variable-reference".

Pascal *symbols*, such as reserved words, operators, and punctuation, are bold-face and are enclosed in circles or ovals, as in the following diagram for the construction of a compound-statement:



Text in a circle or oval represents itself, and is to be written as shown (except that capitalization of letters is not significant). In the diagram above, the semicolon and the words ***begin*** and ***end*** are symbols. The word "statement" refers to a construction that has its own syntax diagram.

A compound-statement consists of the reserved word ***begin***, followed by any number of statements separated by semicolons, followed by the reserved word ***end***. (As will be seen in Chapter 6, a statement may be null; thus ***begin end*** is a valid compound-statement.)

Section 1

Tokens and Constants

1.1	Character Set and Special Symbols	3
1.2	Identifiers	4
1.3	Directives	4
1.4	Numbers	4
1.5	Labels	6
1.6	Character-Strings	6
1.7	Constant-Declarations	7
1.8	Comments	7
1.9	Compiler Options	7

Section 1

Tokens and Constants

Tokens are the smallest meaningful units of text in a Pascal program. Structurally, they correspond to the words and punctuation of an English sentence. The tokens of Pascal are classified into *special symbols*, *identifiers*, *numbers*, *labels*, and *character-strings*.

The text of a Pascal program consists of tokens and *separators*, where a separator is either a *blank* or a *comment*. Two adjacent tokens must be separated by one or more separators if each token is an identifier, number, or word-symbol.

No separators can be embedded within tokens, except in character-strings.

1.1 Character Set and Special Symbols

The character set used by Lightspeed Pascal is the Macintosh character set (see Appendix J).

Letters, digits, hex-digits, and blanks are subsets of the character set:

- The *letters* are those of the English alphabet, *A* through *Z* and *a* through *z*.
- The *digits* are the Arabic numerals *0* through *9*; the *hex-digits* are the Arabic numerals through *9*, the letters *A* through *F*, and the letters *a* through *f*.
- The *blanks* are the space character and the end-of-line character (CR).

Special-symbols and *word-symbols* (sometimes referred to as *reserved words*) are tokens having one or more fixed meanings. The following single characters are special-symbols:

+ - * / = < > [] . , () : ; ^ @ { } \$

The following *character-pairs* are special-symbols:

<> <= >= := .. (* *) (.) .)

Note: The special symbol `(.` is an alternative representation for the special symbol `[` -- they both actually denote the same special symbol. Therefore if `(.` is entered into a Lightspeed Pascal program, `[` will always be redisplayed. The same is true for `.)` and `]`.

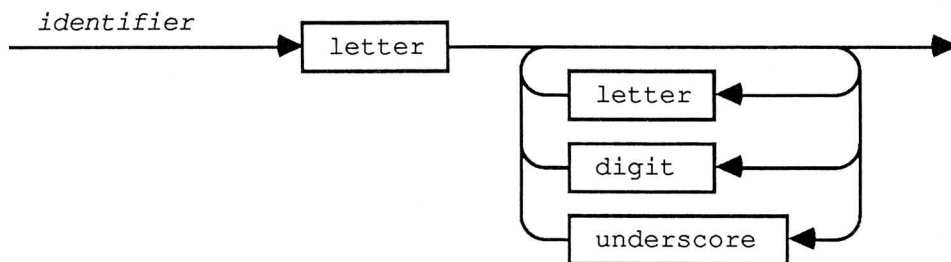
The following are the *word-symbols*:

and	end	interface	procedure	unit
array	file	label	program	until
begin	for	mod	record	uses
case	function	nil	repeat	var
const	goto	not	set	while
div	if	of	string	with
do	implementation	or	then	
downto	in	otherwise	to	
else	inline	packed	type	

Corresponding upper- and lower-case letters are equivalent in word-symbols.

1.2 Identifiers

Identifiers serve to denote constants, types, variables, procedures, functions, programs, and fields in records. Identifiers can be of any length up to 255 characters all of which are significant. Corresponding upper- and lower-case letters are equivalent in identifiers. No identifier can have the same spelling as a word-symbol.



Examples of identifiers:

X *Rome* *gcd* *SUM* *get_byte*

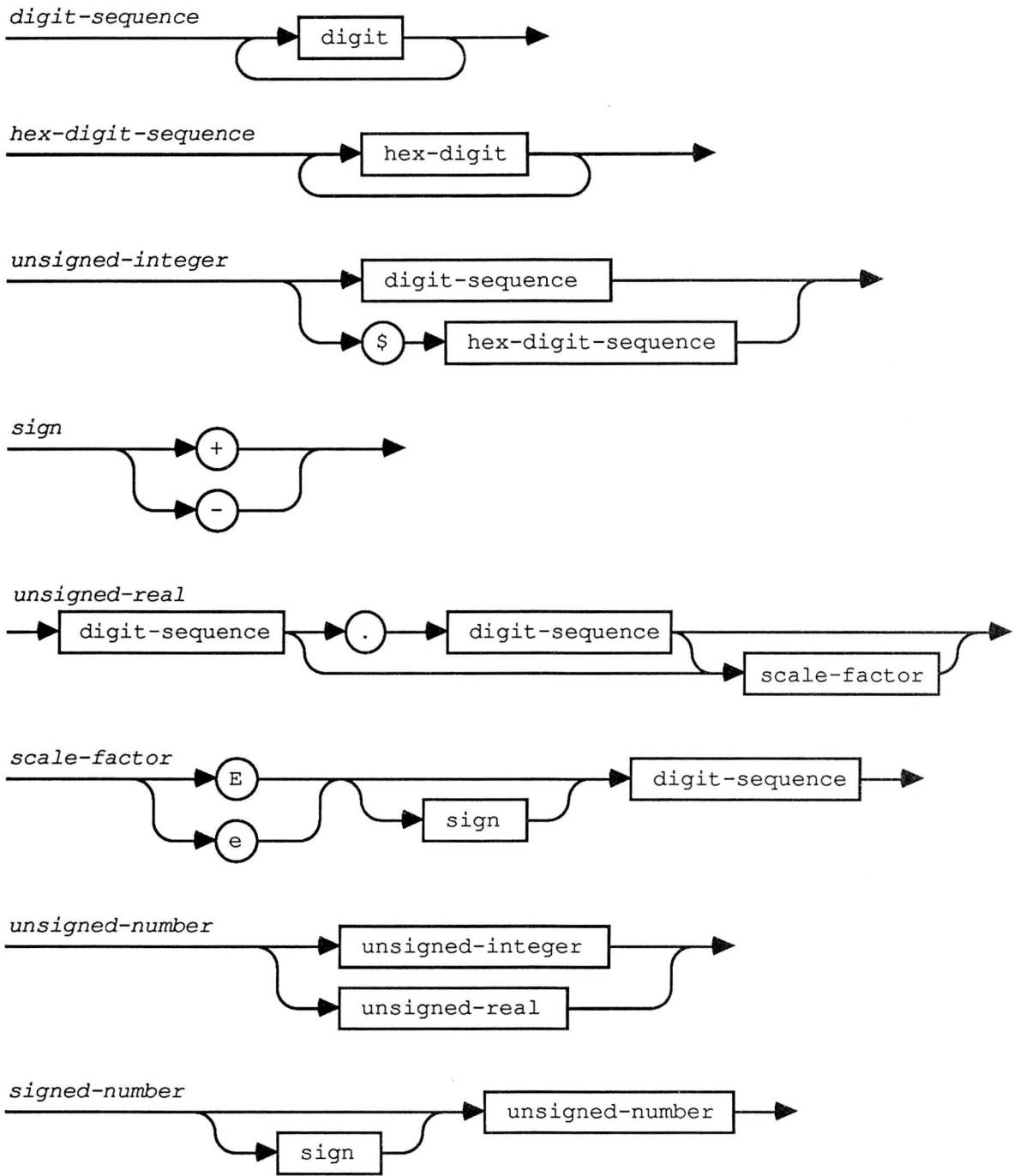
1.3 Directives

Directives are identifiers that have special meanings in specific contexts (see Chapter 7). They can be used as identifiers in all other contexts. For example, the word *forward* is interpreted as a directive if it occurs immediately after a procedure-heading or function-heading, but in any other position it is interpreted as an identifier.

1.4 Numbers

The usual decimal notation is used to represent numbers that are constants of the data types *integer*, *longint*, *real*, *double*, *extended*, and *computational* (see Section

3.1). A hexadecimal integer constant uses the \$ character as a prefix (1-4 hex-digits for *integer*, 5-8 hex-digits for *longint*).



The letter *E* or *e* preceding the scale in an unsigned-real may be read as "times ten to the power of".

Examples of numbers:

1 +100 -0.1 5E-3 87.35e+8 \$A05D

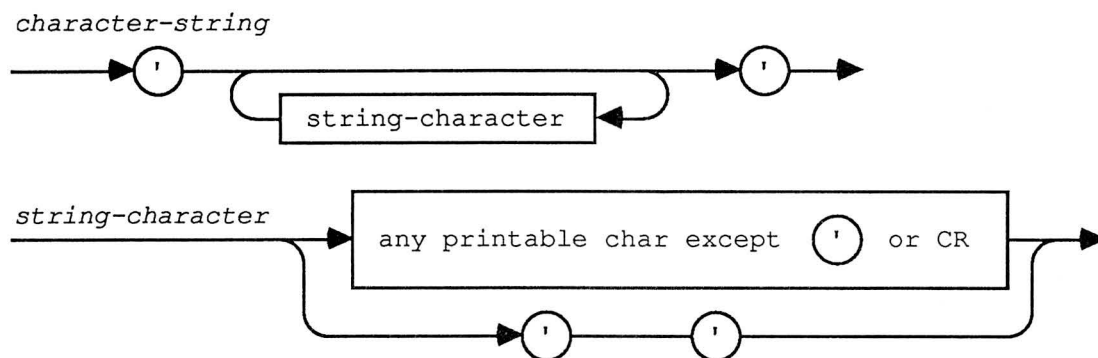
Note that 5E-3 means 5×10^{-3} , and 87.35e+8 means 87.35×10^8 .

1.5 Labels

A label is a digit-sequence whose value is in the range 0 through 9999. Leading zeroes in a label are insignificant, e.g. the labels 1 and 0001 are considered equivalent.

1.6 Character-Strings

A character-string is a sequence of zero or more printing Macintosh characters all on the same line in a program and enclosed by apostrophes. The maximum number of characters that can be in a character-string is 255. A character-string with nothing between the apostrophes denotes a *null-string* value (see Section 3.3). A pair of adjacent apostrophes in a character-string denotes a single apostrophe character.



A character-string represents a value of a *string-type*. As a string-type, a character-string is compatible not only with other string-types, but also *char-types* (see Section 3.1.1.1) and *packed-string-types* (section 3.2.1).

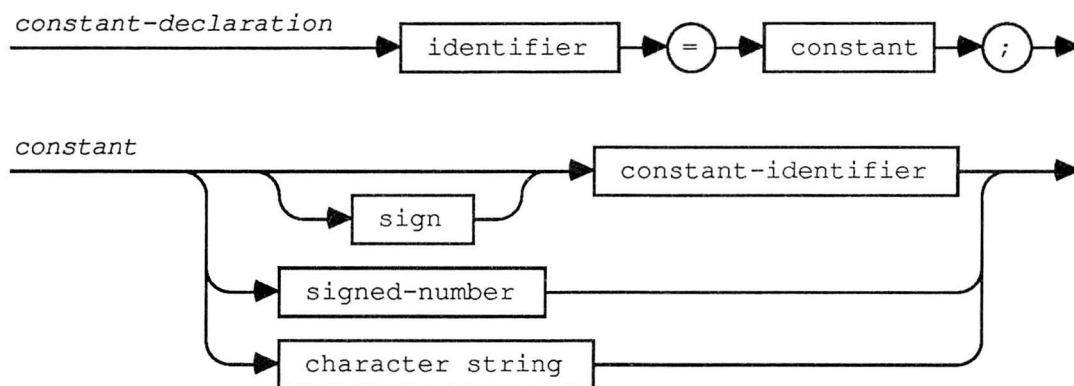
All string-type values have a *length* attribute. In the case of a character-string, the length is fixed; it is equal to the actual number of characters in the string as enclosed within apostrophes. A pair of adjacent apostrophes within a character-string is regarded as a single apostrophe and thus counts as a single character in the string's length.

Examples of character-strings:

'Pascal' 'THIS IS A STRING' 'Don''t worry!'
'A' ';' '''' ''

1.7 Constant-Declarations

A constant-declaration declares an identifier to denote a constant within the block that contains the declaration. A constant-identifier may not be included in its own declaration.



A constant-identifier following a sign must denote a value of type *integer*, *longint*, *real*, *double*, *computational* or *extended* (see Section 3.1).

1.8 Comments

The constructs:

```
{ any text not containing right-brace      }
```

```
(* any text not containing star-right-paren *)
```

are called *comments*.

The substitution of a blank for a comment or a comment for a blank does not alter the meaning of a program. This is to say that a comment, as a separator, may appear anywhere in a program where a blank may appear. However, comments embedded within a line of program text will be moved to the end of the line when redisplayed in the Lightspeed Pascal editing window. Comments on a line by themselves are left as such.

A comment cannot be nested within another comment; an occurrence of the special symbols `}` or `*)` within a comment always terminates the comment.

1.9 Compiler Options

A *compiler option* is a comment that contains the `$` character immediately following the initial comment delimiter; for example, `{ $` or `(* $`. The `$` character is followed immediately by the mnemonic of the compiler option. (see Chapter 13 of the User's Guide for available options). A compiler option will always be redisplayed on a line by itself.

Section 2

Blocks, Scope, and Activations

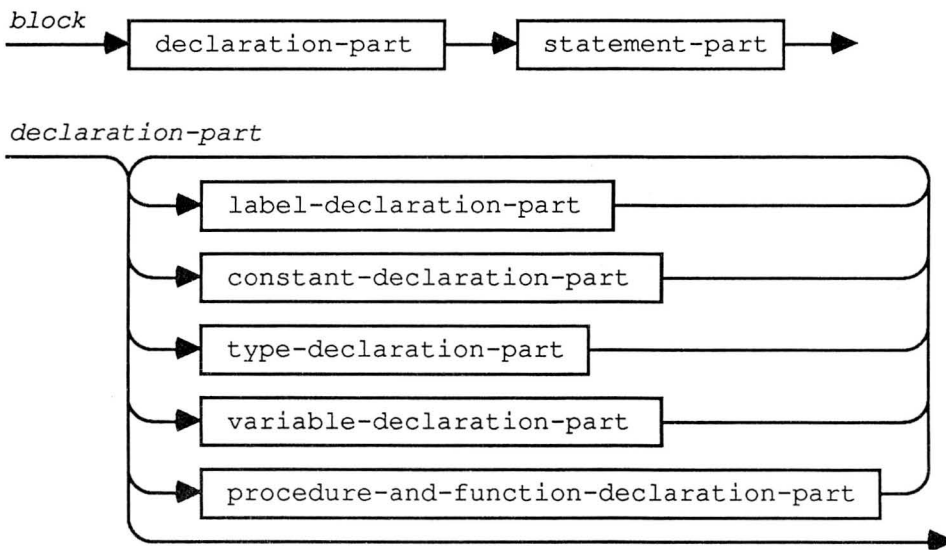
2.1	Definition of a Block	11
2.2	Rules of Scope	12
2.2.1	Scope of a Declaration	12
2.2.2	Redeclaration in an Enclosed Block	13
2.2.3	Position of Declaration within Its Block	13
2.2.4	Redeclaration within a Block	13
2.2.5	Identifiers of Standard Objects	13
2.2.6	Scope of Interface Identifiers	13
2.3	Activations	13

Section 2

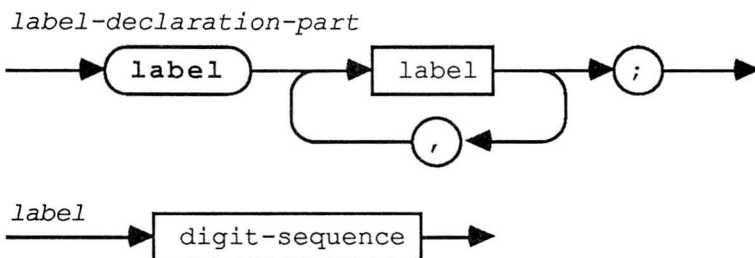
Blocks, Scope, and Activations

2.1 Definition of a Block

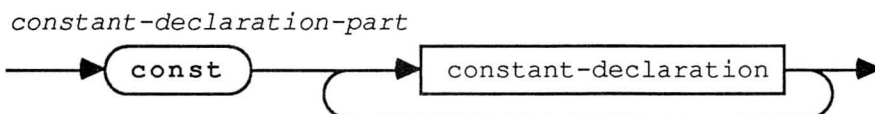
A *block* consists of a *declaration-part* and a *statement-part*. Every block is part of a procedure-declaration, a function-declaration, or a program. All identifiers and labels that are declared in the declaration-part of a block are *local* to that block.



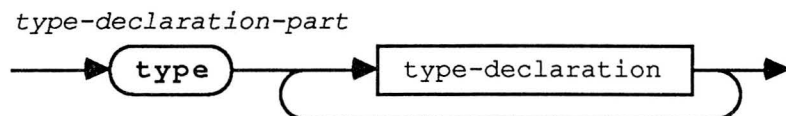
A *label-declaration-part* declares labels (see Section 1.5) that mark statements in the corresponding statement-part. Each label must mark exactly one statement in the statement-part.



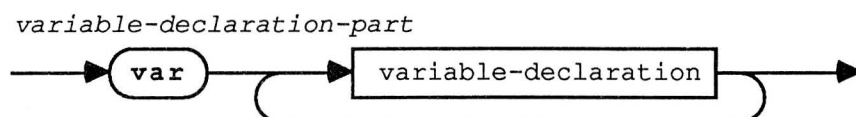
A *constant-declaration-part* contains *constant-declarations* (see Section 1.7) local to the block.



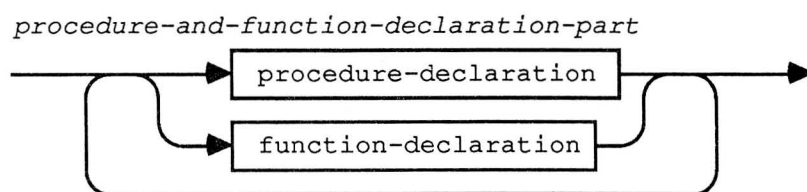
A *type-declaration-part* contains *type-declarations* (see Chapter 3) local to the block.



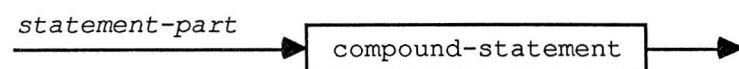
A *variable-declaration-part* contains *variable-declarations* (see Chapter 4) local to the block.



A *procedure-and-function-declaration-part* contains *procedure* and *function-declarations* (see Chapter 7) local to the block.



The *statement-part* specifies the statements or algorithmic actions (see Chapter 6) to be executed upon an *activation* (see Section 2.3) of the block.



2.2 Rules of Scope

2.2.1 Scope of a Declaration

The appearance of an identifier or label in a declaration declares the identifier or label. All other applied occurrences of the identifier or label must be within the *scope* of this declaration.

The scope of a declaration is the block that contains the declaration, and all blocks enclosed by that block except as explained in Section 2.2.2 and 2.2.4 below. (See also Section 8.3 for scope rules for Units.)

2.2.2 Redeclaration in an Enclosed Block

Suppose that *outer* is a block, and that *inner* is another block declared within *outer*. If an identifier declared in block *outer* has the same spelling as an identifier declared in block *inner*, then block *inner* and all blocks enclosed by *inner* are excluded from the scope of the declaration in block *outer*.

2.2.3 Position of Declaration within Its Block

The declaration of an identifier or label must precede all applied occurrences of that identifier or label in the program text -- i.e., identifiers and labels cannot be used until they are declared.

There is one exception to this rule: in a type-declaration-part, the domain-type of a pointer-type (see Section 3.4) can be an identifier that has not yet been declared. In this case, the identifier must be declared somewhere in the same type-declaration-part as the pointer-type.

2.2.4 Redeclaration within a Block

An identifier or label cannot be declared more than once within a block, unless it is declared within a contained block, or it is in a record's field-list.

A field-identifier (see Sections 3.2.2 and 4.3.2) is declared within a record-type. It is meaningful only in combination with a reference to a variable of that record-type. Therefore, a field-identifier can be declared within the same block as another identifier with the same spelling, as long as it has not been declared previously in the same field-list. An identifier that has been declared can be used again as a field-identifier in the same block.

2.2.5 Identifiers of Standard Objects

Lightspeed Pascal provides a set of standard (predeclared) constants, types, procedures, and functions that behave as if they were declared in a block that contains the entire program. In addition, there are two standard file variables, *input* and *output*, that (if used) are "declared" in the program block itself (see Section 9.4).

2.2.6 Scope of Interface Identifiers

Each interface-part or program with a *uses*-clause (see Section 8.4) is *supplied* with the identifiers associated with the unit given in the *uses*-clause. This means that these identifiers behave as though they were declared directly in each interface-part or program with the *uses*-clause.

2.3 Activations

The execution of a block is referred to as an *activation* of the block. At any given time, a block normally has either no activations (if it is not currently being executed) or one activation (if it is). It is, however, possible for a block to have multiple activations if it is *recursive* or if it is *mutually recursive* with one or more other procedures or functions. A typical example of a recursive function is:

```

function factorial( n : integer ) : integer;
begin
    if n<=1 then
        factorial := 1
    else
        factorial := n * factorial(n-1)
    end;

```

Thus, the execution of *factorial*(5) would lead to 5 activations of *factorial* as follows:

```

factorial(5) = 5 * factorial(4)
              = 5 * 4 * factorial(3)
              = 5 * 4 * 3 * factorial(2)
              = 5 * 4 * 3 * 2 * factorial(1)
              = 5 * 4 * 3 * 2 * 1

```

Factorial repeatedly calls itself, creating new activations, until the parameter *n* that is passed is less than or equal to one. The last activation then *unwinds* itself by passing back a result and terminating the activation. The next to last activation then performs the multiplication with the result, passes back its result, and terminates its activation, and so on.

Every activation is given, in effect, its own *copy* of every parameter and variable declared to be local to the block being activated. Thus, each activation of *factorial* above has its own copy of its parameter, which is named *n* in all activations. Because each activation has its own copy of all locally declared entities, it does not disturb the local entities of any previous activations. Section 7.3.3 gives a very detailed example of this.

The activation of the program-block also causes creation of the variables local to each used unit's interface-part and implementation-part (see Section 8.3). This means that there is only one copy of each unit's local variables and that they exist as long as the program is executing.

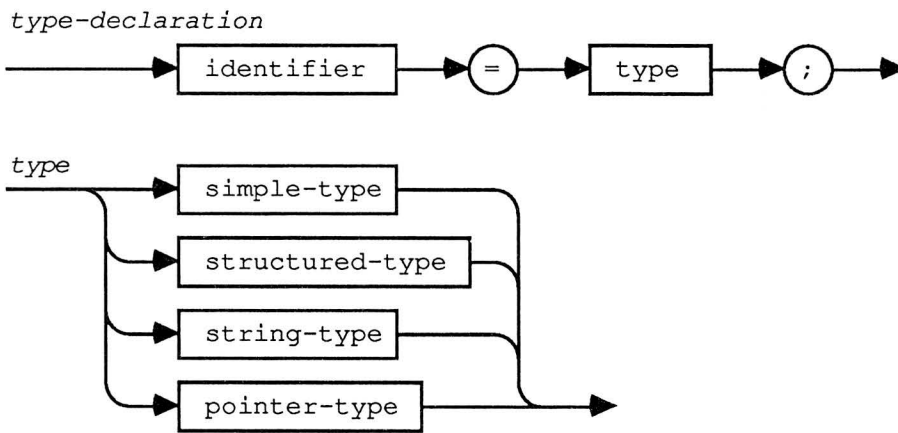
Note: The value of a variable declared within a particular block is undefined for each new activation of the block. Likewise, the value of every component of a structured-type variable (see Section 3.2) is initially undefined for each new activation. The value of a structured-type variable remains undefined until it has no components whose values are undefined.

Section 3 Types

3.1 Simple-Types	18
3.1.1 Ordinal Types	18
3.1.1.1 Standard Ordinal-Types	19
3.1.1.2 Enumerated-Types	20
3.1.1.3 Subrange-Types	21
3.1.2 Real-Types	21
3.2 Structured-Types	23
3.2.1 Array-Types	23
3.2.2 Record-Types	24
3.2.3 Set-Types	26
3.2.4 File-Types	27
3.3 String-Types	27
3.4 Pointer-Types	28
3.5 Identical and Compatible Types	28
3.5.1 Type Identity	28
3.5.2 Compatibility of Types	29
3.5.3 Assignment-Compatibility	30
3.6 The Type-Declaration-Part	31

Section 3 Types

A *type* is used in declaring variables, as well as in declaring other types. The type of a variable determines the set of values that the variable can assume and the operations that can be performed upon it. A *type-declaration* associates an identifier with a type.



The occurrence of an identifier on the left-hand side of a type-declaration declares it as a type-identifier for the block in which the type-declaration occurs. A type-identifier may not be included in its own declaration, except for pointer-types (see Sections 2.2.3 and 3.4).

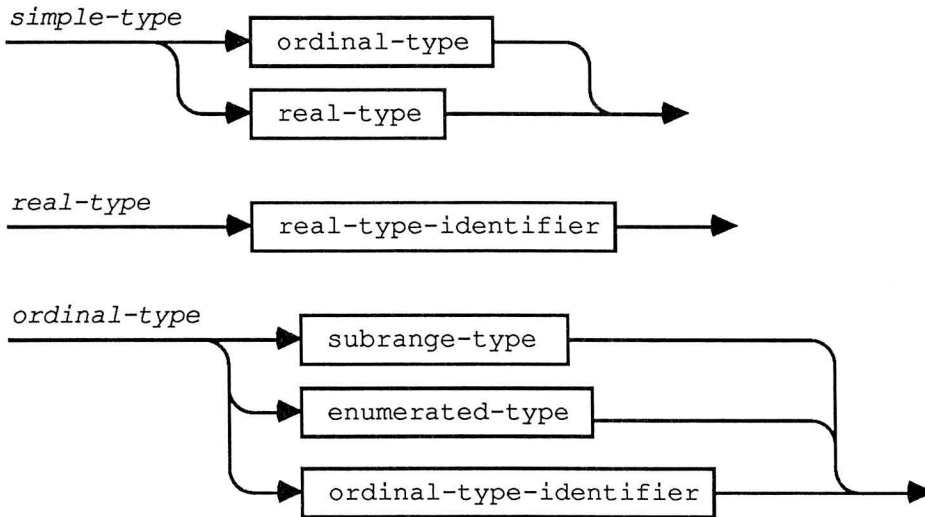
To help clarify the syntax description with some semantic hints, the following terms are used to distinguish identifiers according to what they denote. Syntactically, all of them simply mean an identifier:

- simple-type-identifier
- structured-type-identifier
- pointer-type-identifier
- ordinal-type-identifier
- integer-type-identifier
- real-type-identifier
- string-type-identifier

In other words, a simple-type-identifier is any identifier that is declared to denote a simple-type, a structured-type-identifier is any identifier that is declared to denote a structured-type, and so forth. A simple-type-identifier can be the identifier of a standard simple-type such as *integer*, *boolean*, etc.

3.1 Simple-Types

All the simple-types define ordered sets of values.



An integer-type-identifier is one of the standard identifiers *integer* or *longint*. Constant integer-type values can be denoted as described in Sections 1.4 and 1.7.

A real-type-identifier is one of the standard identifiers *real*, *double*, *extended*, or *computational*. Constant real-type values can be denoted as described in Sections 1.4 and 1.7.

3.1.1 Ordinal-Types

Ordinal-types are a subset of the simple-types that have the following special characteristics:

- The possible values of an ordinal-type are an ordered set and every value has an *ordinality*, which is an integral value. Except for integer-types, the first value of every ordinal-type has ordinality 0, the next has ordinality 1, etc. For integer-types, the ordinality of a value is the value itself. Every value of an ordinal-type except the first has a *predecessor* based on the ordering of the type, and every value of an ordinal-type except the last has a *successor* based on the ordering of the type.
- The standard function *ord* (see Section 10.4.1) can be applied to any value of an ordinal-type, and it returns the ordinality of the value.
- The standard function *pred* (see Section 10.4.4) can be applied to any value of an ordinal-type, and it returns the predecessor of the value.
- The standard function *succ* (see Section 10.4.3) can be applied to any value of an ordinal-type, and it returns the successor of the value.

The application of *pred* to the first value of an ordinal-type is an error. Likewise, the application of *succ* to the last value of an ordinal-type is an error.

All simple-types except the real-types are ordinal-types.

There are four standard ordinal-types denoted by the standard identifiers:

```
integer
longint
char
boolean
```

Note that in addition to the standard ordinal-types, the enumerated-types and subrange-types are ordinal-types.

3.1.1.1 Standard Ordinal-Types

Integer The integer-type *integer* has a set of values that are a subset of the whole numbers. The standard *integer* constant *maxint* is defined to be $2^{15}-1$, i.e. 32,767. A variable of type *integer* can have any value in the range $-maxint..maxint$. Values of type *integer* are 16-bit, signed, 2's-complement numbers.

Longint The integer-type *longint* is a type that has a set of values that are also a subset of the whole numbers, a somewhat larger subset than those of *integer*. The standard *longint* constant *maxlongint* is defined to be $2^{31}-1$, i.e. 2,147,483,647. A variable of type *longint* can have any value in the range $-maxlongint..maxlongint$. Values of type *longint* are 32-bit, signed, 2's-complement numbers.

There are several *arithmetic operators* that may be used to perform arithmetic with integer-type values. All arithmetic with just integer-type *integer* operands yields results of type *integer*. When one or both operands are of integer-type *longint*, the result is always of type *longint*. A *longint* value may always be used where an *integer* value is required provided that the value falls within the range $-maxint..maxint$.

Boolean The ordinal-type *boolean* is an enumerated-type (see Section 3.1.1.2) defined as:

```
type boolean = ( false, true );
```

As a consequence of *boolean* being an enumerated-type, the following relationships hold:

```
false      < true
ord(false) = 0
ord(true)  = 1
succ(false) = true
pred(true) = false
```

Char The ordinal-type *char* has a set of values that are characters. The ordering of the values is defined by the ordering of the Macintosh character set (see Appendix J). The function-call *ord(c)*, where *c* is a *char* value, returns the ordinality of *c* (see Section 10.4.1).

A character-string of *length* 1 may be used to denote a constant *char* value, provided that the character is a printable character. Any value of type *char* may be generated via the standard function *chr* (see Section 10.4.2).

Note: The redeclaration of a standard type-identifier does not affect the operand-types, parameter-types, or result-types of certain standard operators, procedures, and functions declared to be that standard type (see Section 5.1 and Chapter 10). Neither does it affect the type of any literal token (such as a *number*) declared to be of that type (see Chapter 1).

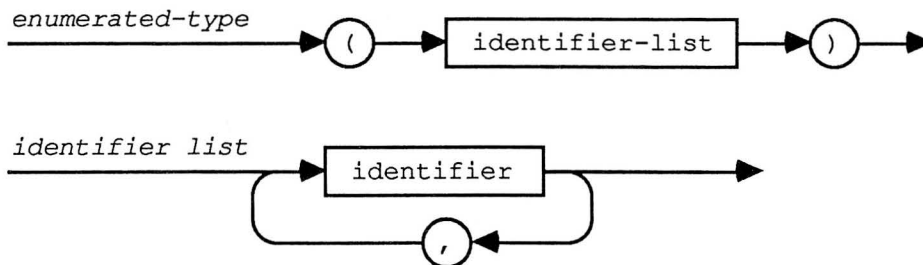
The redeclaration of the standard constant-identifiers *maxint* and/or *maxlongint* has no effect on the set of possible values for the integer-types.

There are several contexts in which an expression is required to be of the standard type *boolean* (see Chapter 6). A redeclaration of the standard type-identifier *boolean* does not alter this requirement.

However, the redeclaration of the standard enumerated-constant identifiers *false* and *true* will affect the value of these identifiers.

3.1.1.2 Enumerated-Types

An enumerated-type has an ordered set of values defined by listing the identifiers that denote these values. The ordering of these values is determined by the sequence in which the identifiers are listed.



The occurrence of an identifier within the identifier-list of an enumerated-type declares it as an enumerated-constant for the block in which the enumerated-type is declared. The type of this constant is the enumerated-type in which it is declared.

The ordinality of an enumerated-constant is its position in the identifier-list in which it is declared, where the ordinality of the first enumerated-constant in the list is always 0. The ordinality of a value of an enumerated-type is the ordinality of the enumerated-constant with the same value.

When the *ord* function (see Section 10.4.1) is applied to a value *v* of an enumerated-type, it returns an integer-type value that is the ordinality of *v*.

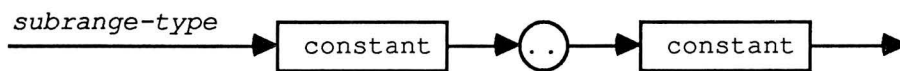
Examples of enumerated-types:

```
color = ( red, yellow, green, blue )
suit  = ( club, diamond, heart, spade )
maritalStatus = ( married, divorced, widowed, single )
```

Given these declarations, *yellow* is an enumerated-constant of type *color* with ordinality 1, *spade* is an enumerated-constant of type *suit* with ordinality 3, and so forth.

3.1.1.3 Subrange-Types

A subrange-type has a subset of the values of some ordinal-type that lie within a certain range. The syntax for a subrange-type is



Both constants in a subrange-type must be of an ordinal-type and both must be of the same ordinal-type. The value of an integer-type constant must fall in the range $-maxint..maxint$, i.e. it cannot be of type *longint*. For all subrange-types of the form $a..b$, a must be less than or equal to b . The ordinal-type of a and b is referred to as the *host-type* of the subrange-type. The values of a subrange-type $a..b$ are those values of its host-type whose ordinalities lie between the ordinalities of a and b inclusive.

Examples of subrange-types:

```
1..100
-10..+10
red..green
```

A variable of subrange-type possesses all the properties of variables of the host-type, with the restriction that its value must always be one of the values in the range defined by the subrange-type.

3.1.2 Real-Types

The real-types have sets of values that are subsets of the real numbers; in particular those subsets of the real numbers that may be represented with a floating-point notation using a fixed number of digits. In general, a floating-point notation of a value n is comprised of a set of three values m , b , and e such that

$$m \times b^e \approx n$$

A real-type uses a floating point notation where b is always 2, and where m and e are integral values that lie in a range that depends on the particular real-type. The range of values that m and e can have determine the *range* and *precision* of the real-type. More detailed information on the representation of real-type values can be found in Appendix D.

Doing arithmetic with real-type values can lead to results that cannot even be approximated with a floating-point notation. For instance, the division of one by zero is nominally ∞ . Other results make even less sense, such as dividing zero by zero. There are, in fact, special real-type values to

represent such results. Normally, however, the generation of such a result by a program is an error and results in the premature termination of the execution of the program. Appendix D describes alternative methods for dealing with such results.

There are four standard real-types: *real*, *double*, *extended*, and *computational*. No other real-types are possible.

The approximate range of positive values representable with the real-types *real*, *double*, and *extended* as well as their precision are as follows:

<u>real-type</u>	<u>range</u>	<u>decimal digits</u>
<i>real</i>	1.5×10^{-45} to 3.4×10^{38}	7-8
<i>double</i>	5.0×10^{-324} to 1.7×10^{308}	15-16
<i>extended</i>	1.9×10^{-4951} to 1.1×10^{4932}	19-20

Any negative value whose absolute value lies within these ranges is representable with these real-types.

All real-type operands are converted to *extended* before any arithmetic is performed on them, and the results of such arithmetic are always of type *extended*. An *extended* value may always be used where a *real* or *double* value is required provided that the value falls within the range of values for *real* or *double* respectively.

The real-type *computational* is a special real-type where *e* is always zero, i.e. only integral values may be represented with *computational*. Values of type *computational* must lie in the (exact) range $-2^{63}+1$ to $2^{63}-1$, which is approximately -9.2×10^{18} to 9.2×10^{18} .

All *computational* operands are converted to *extended* before any arithmetic is performed on them, and the results of such arithmetic are always of type *extended*. An *extended* value may always be used where a *computational* value is required provided that the value, when rounded to an integral value, falls within the range of values for *computational*.

The type *computational* is intended for those applications where precise, fixed-point decimal values are required. No explicit decimal point is ever assumed for a *computational* value, but one can be implicitly assumed by the application. For instance, one can define

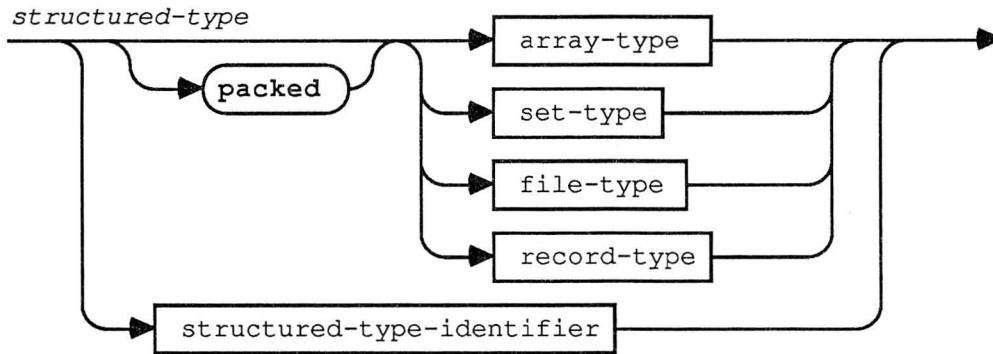
```
type cents = computational;
```

and perform calculations on values of type *cents* that may also be interpreted as calculations on dollar values with an implied decimal point to the left of the second to last decimal digit. A special form of the standard procedures *write* and *writeln* (see Section 9.4.3.5) may be used to output a *computational* value with a decimal point inserted between any two decimal digits in the value.

Note: Users of SANE (see Appendix D) should be aware that the SANE data types *single* and *comp* are equivalent to the Pascal real-types *real* and *computational* respectively. All of these names are accepted by Lightspeed Pascal.

3.2 Structured-Types

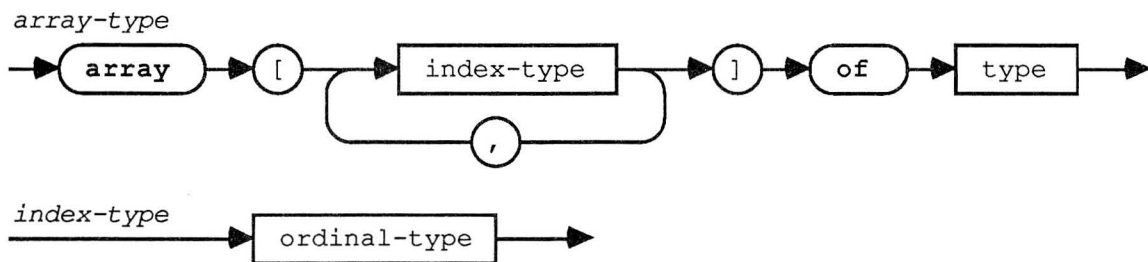
A structured-type is characterized by the kind of structuring it embodies and by the type(s) of the components subject to such structuring. The type of a component may itself be structured, in which case the resulting structured-type exhibits more than one level of structuring. There is no inherent limit on the number of levels to which types can be structured other than the amount of memory available.



The use of the word ***packed*** in the declaration of a structured-type indicates that storage organization of all values of that type should be compressed to economize storage, even if this causes the access of a component of a variable of this type to be less efficient.

3.2.1 Array-Types

An array-type is a linear array (or vector) of components that are all of one type, called the *component-type* of the array.



The type that follows the word ***of*** is the component-type of the array. The number of elements is determined by the *index-type* of the array, which must be an ordinal-type.

If the component-type of an array-type is also an array-type, the result can be regarded as a single multi-dimensional array. An equivalent, shorthand notation may be used to declare multi-dimensional arrays that entails declaring a single array with multiple index-types. For example, the type

```
array[boolean] of array[1..10] of array[size] of real
```

is equivalent to the type

```
array[ boolean, 1..10, size ] of real
```


where "equivalent" means that they will be interpreted in the same way.

Examples of array-types:

```
array[1..100] of real
packed array[color] of boolean
```

A component of an array can be accessed by referencing the array and supplying one or more indices (see Section 4.3.1).

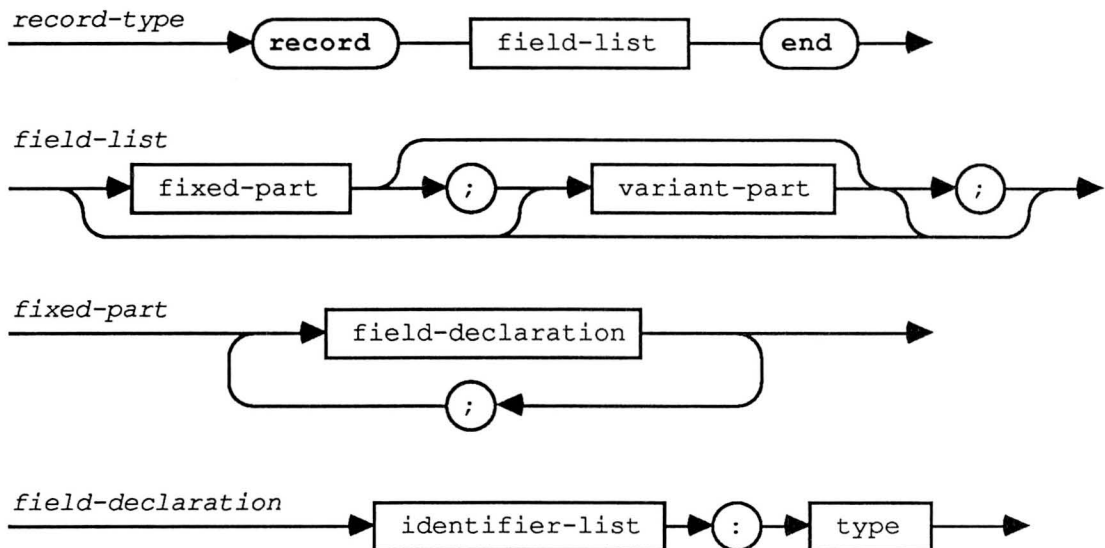
An array-type of the form

```
packed array[1..n] of char
```

is referred to as a *packed-string-type* with n components (in American National Standard Pascal these are called *string-types*; they are called *packed-string-types* here to avoid confusion with Lightspeed Pascal's *string-types* (see Section 3.3). A packed-string-type has certain properties not shared by other array-types or structured-types (see Sections 3.5 and 5.1.5).

3.2.2 Record-Types

A record-type consists of a fixed number of components called *fields*, each of which may be a different type. For each component, the record-type specifies the type of the field and an identifier that denotes it.

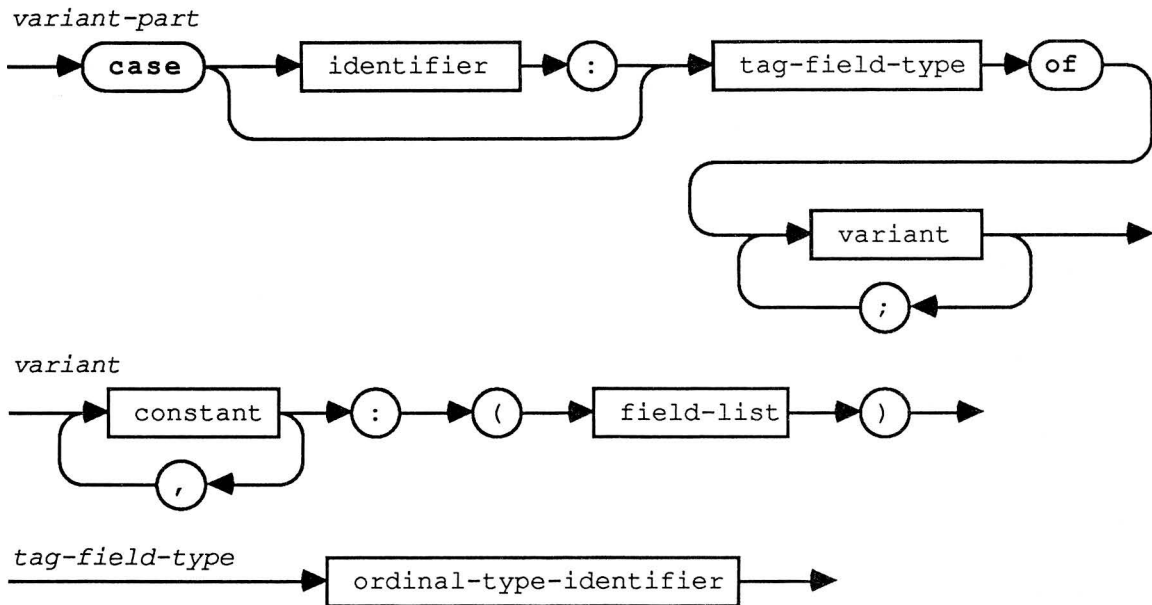


The fixed-part of a record-type specifies a field-list that is always accessible in a variable of the record-type, giving an identifier and a type for each field. Each of these fields contains data that is always accessed in the same way (see Section 4.3.2).

Example of a record-type:

```
record
  year: integer;
  month: 1..12;
  day: 1..31
end
```

A variant-part consists of alternative field-lists of which only one is accessible at a given time. Each alternative field-list is called a *variant*.



Each variant is preceded by one or more constants. All of the constants must be distinct and must be of an ordinal-type that is compatible with the tag-type (see Section 3.5).

The variant-part allows for an optional identifier that denotes a *tag-field*. If a tag-field is present, it is considered a field of the fixed-part. The value of the tag-field indicates which variant is *active*, and thus which variant's fields are accessible at a given time. If there is no tag-field explicitly given, then all fields in all variants are always accessible, i.e. the active variant is the one containing the field most recently referenced.

like Fortran equivalence?

It is an error to alter the value of a tag-field while a reference to a field of the active variant exists. Whenever the value of the tag-field changes, the values of all of the fields in the newly active variant are undefined. When the value of the tag-field is undefined, the values of all fields in all variants of the corresponding variant-part are undefined.

Note: All of the variants of a variant-part share the same region of memory within a record variable. This is because only one particular variant is ever in use at a time (see also Section 10.1).

It is not uncommon for Pascal programmers to use a variant-part with no explicit tag-field as a means of converting a value of one type to a value of another type, namely by assigning a value into a field of one variant and referencing the corresponding field in another variant, e.g.,

```

var
  r: record
    case boolean of
      false: ( CharVal: char );
      true:  ( IntVal: integer )
    end;
  ...
  r.CharVal := 'W';
  ...
  c := r.IntVal;
  ...

```

However, doing this assumes a knowledge of the underlying representation of variables in memory, and is therefore not recommended. This sort of "type coercion" should be performed using the predefined functions *chr* and *ord*, or with type casts (see Section 5.4) which are guaranteed to do what you would expect.

Examples of record-types with variants:

```

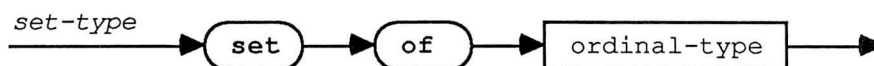
record
  name, firstName: string[80];
  age: 0..99;
  case married: boolean of
    true:  (spousesName: string[80]);
    false: ()
  end
end

record
  x,y: real;
  area: real;
  case s: shape of
    triangle: (side: real;
               inclination, angle1, angle2: angle);
    rectangle: (side1, side2 : real;
               skew, angle3: angle);
    circle:   (diameter: real)
  end
end

```

3.2.3 Set-Types

A set-type has a range of values that is the powerset of some ordinal-type, called the *base-type*. In other words, each possible value of a set-type is a subset of the possible values of the base-type.

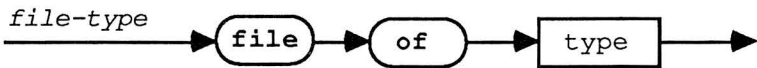


Operators applicable to sets are specified in Section 5.1.4. Section 5.3 shows how set values are denoted in Pascal.

The empty set (see Section 5.3) is a possible value of every set-type.

3.2.4 File-Types

A file-type is a structured-type consisting of a linear sequence of components that are all of one type, the *component-type*. The component-type may be any type that is not a file-type or a structured-type that contains a file-type component at any level of structuring. The number of components in a file-type variable is not fixed by the file-type declaration.



The standard file-type *text* denotes a (packed) file of characters organized into lines. Files of type *text* are supported by the specialized I/O procedures discussed in Section 9.4.

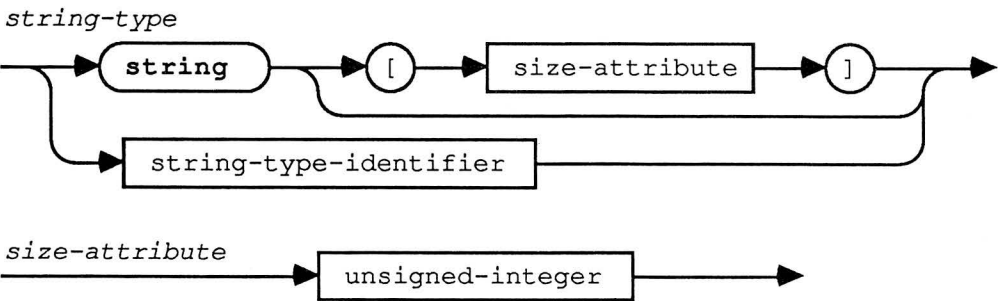
Sections 4.3.3 and Chapter 9 discuss methods of accessing file components.

3.3 String-Types

A string-type value is a sequence of characters that has a dynamic *length* attribute. The length is the actual number of characters in the sequence at any time during program execution.

A string-type has a static *size* attribute that is an integral value in the range from 1 to 255. The size is a maximum limit on the length of any value of this type. If an explicit size attribute is not given for a string-type, then it is given a size of 255 by default.

The current value of the length attribute of a string-type value is returned by the standard function *length* (see Section 10.6). A *null-string* is a string-type value with a length of zero.



The ordering relationship between any two string values is determined by lexical comparison based on the ordering relationship between character values in corresponding positions in the two strings. When the two strings are of unequal lengths, each character in the longer string that does not correspond to a character in the shorter one compares "higher"; thus the string value *'attribute'* is *greater than* the value *'at'*. Two strings must always have the same lengths to be equal; *'X '* (X followed by a space) is always greater than *'X'* (just X).

A null-string is only equal to another null-string and is less than every other possible string value.

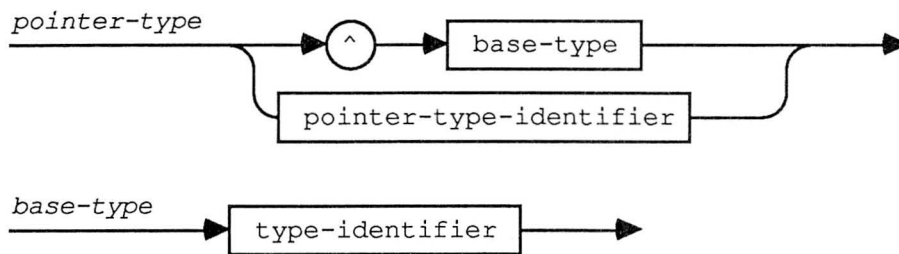
Do not confuse the size with the length.

There are aspects of string-types that make it seem both a simple-type and a structured-type at the same time; it can therefore be considered neither. However, as explained in Section 4.3.1, individual characters in a string can be accessed as if they were components of an array. Operators applicable to strings are specified in Section 5.1.5. Standard procedures and functions for manipulating strings are described in Section 10.5.

3.4 Pointer-Types

A pointer-type defines a set of values that point to *dynamic-variables* of a specified type called the *base-type*.

Pointer values are created by the standard procedure *new* (see Section 10.1.1), by the @ operator (see Section 5.1.6), and by the standard procedure *pointer* (see Section 10.2.6).



The base-type may be an identifier that has not yet been declared. In this case, it must be declared somewhere in the same type-declaration-part as the pointer-type (see Section 2.2.3).

The special symbol ***nil*** represents a standard pointer-valued constant that is a possible value of every pointer type. Conceptually, ***nil*** is a pointer that does not point to anything.

Section 4.3.4 discusses the syntax for referencing the object pointed to by a pointer variable.

3.5 Identical and Compatible Types

Two types may or may not be *identical*, and identity is required in some contexts. Other times, even if not identical, two types need only be *compatible*, and other times *assignment-compatibility* is required.

3.5.1 Type Identity

Identical types are required *only* in the following contexts:

- For variable parameters, the actual and formal parameters must be identical types (see Section 7.3.2).
- The result types of actual and formal functional parameters must be identical (see Section 7.3.4).
- Value and variable parameters within parameter-lists of actual and formal procedural or functional parameters must be identical types (see Section 7.3.5).

Two types, T_1 and T_2 , are *identical* if one of the following is true:

- T_1 and T_2 are the same type-identifier.
- T_1 is declared to be equivalent to a type identical to T_2 .

What the latter, somewhat circular statement implies is that T_1 need not be declared directly to be equivalent to T_2 ; thus the type-declarations

```
T1 = integer;  
T2 = T1;  
T3 = integer;  
T4 = T2;
```

result in T_1 , T_2 , T_3 , T_4 , and *integer* all being identical types. However, the type-declarations

```
T5 = set of integer;  
T6 = set of integer;
```

do not result in T_5 and T_6 being identical, since **set of integer** is not a type-identifier. Finally, note that two variables declared in the same declaration, as in

```
V1, V2: set of integer
```

are of identical type. However, if the declarations are separate then the above definitions apply. The declarations

```
V1: set of integer;  
V2: set of integer;  
V3: integer;  
V4: integer;
```

result in V_3 and V_4 being of identical type, but not V_1 and V_2 .

3.5.2 Compatibility of Types

Compatibility between two types is sometimes required. Specific instances where type compatibility is required are noted elsewhere in this manual. Type compatibility is, more importantly, often a *precondition* of *assignment-compatibility*.

Two types are *compatible* if any of the following are true:

- Both types are identical.
- Both types are real-types.

- Both types are integer-types.
- One type is a subrange of the other.
- Both types are subranges of identical host-types.
- Both types are set-types with compatible base-types.
- Both types are packed-string-types with the same number of components.
- One type is a string-type and the other is a string-type, packed-string-type, or char-type.

3.5.3 Assignment-Compatibility

Assignment-compatibility is required whenever a value is assigned to something, either explicitly (as in an assignment-statement) or implicitly (as in passing value parameters).

A value V_2 of type T_2 is assignment-compatible with a variable V_1 of type T_1 (i.e. $V_1 := V_2$ is permissible) if any of the following are true:

- T_1 and T_2 are identical types and neither is a file-type or a structured-type that contains a file-type component at any level of structuring.
- T_1 and T_2 are real-types and the value of type T_2 is within the range of possible values of T_1 .
- T_1 is a real-type and T_2 is an integer-type.
- T_1 and T_2 are compatible ordinal-types, and the value of type T_2 is within the range of possible values of T_1 .
- T_1 and T_2 are compatible set-types, and all the members of the value of type T_2 are within the range of possible values of the base-type of T_1 .
- T_1 and T_2 are compatible packed-string-types.
- T_1 is a char-type and the value of type T_2 is a string-type and has a length of 1.
- T_1 is a packed-string-type with n components and the value of type T_2 is a string-type and has a length of n .
- T_1 is a string-type with a size of n and the value of type T_2 is a string-type and has a length less than or equal to n .

- T_1 is a string-type with a size of n and the value of type T_2 is a packed-string-type with a number of components less than or equal to n .
- T_1 is a string-type and T_2 is a char-type.

It is an error if assignment-compatibility is required and none of the above is true.

3.6 The Type-Declaration Part

Any program, procedure, or function that declares types contains a type-declaration-part, as shown in Chapter 2.

Example of a type-declaration-part:

```
type
  count = integer;
  range = integer;
  color = (red, yellow, green, blue);
  sex   = (male, female);
  year  = 1900..1999;
  shape = (triangle, rectangle, circle);
  card  = array[1..80] of char;
  str   = string[80];
  polar = record
    r: real;
    theta: angle
  end;

person = ^personDetails;
personDetails = record
  name, firstName: str;
  age: integer;
  married: boolean;
  father, child, sibling: person;
  case s: sex of
    male: (enlisted, bearded: boolean);
    female: (pregnant: boolean)
  end;
people = file of personDetails;
infile = file of integer
```

In the above example *count*, *range*, and *integer* denote identical types. The type *year* is compatible and assignment-compatible with, but not identical to, the types *range*, *count*, and *integer*.

Section 4

Variables

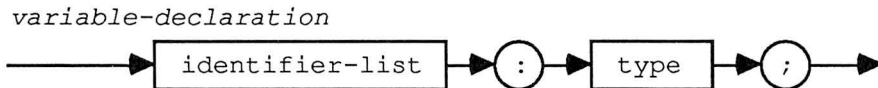
4.1	Variable-Declarations	35
4.2	Variable-References	35
4.3	Qualifiers	36
4.3.1	Arrays, Strings, and Indexes	37
4.3.2	Records and Field-Designators	38
4.3.3	File-Buffers	38
4.3.4	Pointers and Dynamic-Variables	39

Section 4

Variables

4.1 Variable-Declarations

A variable-declaration consists of a list of identifiers denoting new variables, followed by their type.



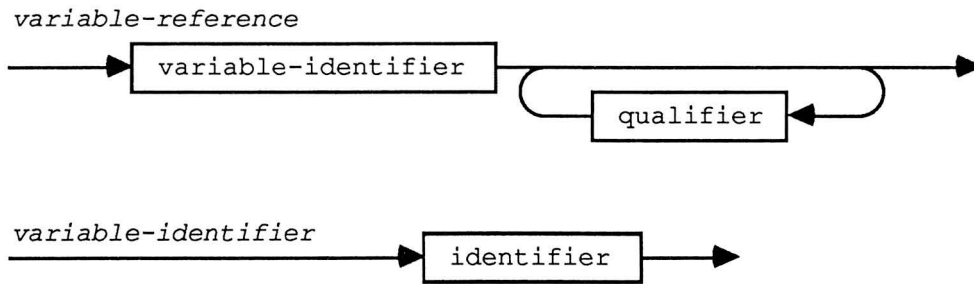
The occurrence of an identifier within the identifier-list of a variable-declaration declares it as a variable-identifier for the block in which the declaration occurs. The variable can then be referenced throughout the remainder of that block, except as specified in Section 2.2.2.

Examples of variable-declarations:

```
x,y,z: real;
i,j: integer;
k: 0..9;
p,q,r: boolean;
operator: (plus, minus, times);
a: array[0..63] of real;
c: color;
f: file of char;
hue1,hue2: set of color;
p1,p2: person;
m,m1,m2: array[1..10,1..10] of real;
coord: polar;
pooltape: array[1..4] of tape
```

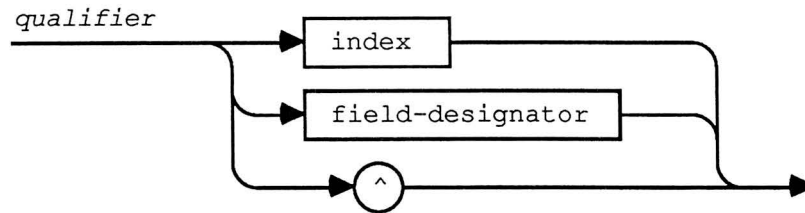
4.2 Variable-References

A variable-reference denotes either an entire variable, a component of a structured or string-type variable, a dynamic-variable pointed to by a pointer-type variable, or the file-buffer of a file-type variable.



4.3 Qualifiers

As shown above, a variable-reference is a variable-identifier followed by zero or more *qualifiers*. Each qualifier modifies the meaning of the variable-reference.



As an example, an array identifier with no qualifier is a reference to the entire array-variable:

```
xResults
```

If the array-identifier is followed by an index, this denotes a specific component of the array:

```
xResults[current+1]
```

If the component is a record, the index may be followed by a field-designator; in this case the variable-reference denotes a specific field within a specific array component.

```
xResults[current+1].link
```

If the field is a pointer, the field-designator may be followed by the symbol `^` to distinguish between the pointer field and the dynamic-variable being pointed to:

```
xResults[current+1].link^
```

If the object of the pointer is an array, another index can be added to denote a component of this array:

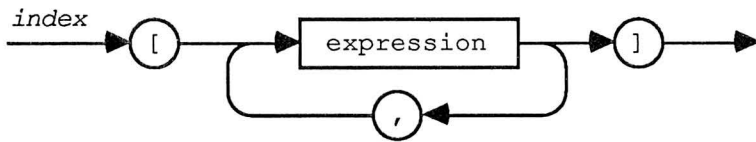
```
xResults[current+1].link^[i]
```

and so on...

4.3.1 Arrays, Strings, and Indexes

A specific component of an array variable is denoted by a variable-reference that refers to the array variable, followed by an index that specifies the component.

A specific character within a string variable is denoted by a variable-reference that refers to the string variable, followed by an index that specifies the character position.



Examples of indexed arrays:

```
m[i, j]
a[i+j]
```

Each expression in the index selects a component in the corresponding dimension of the array. The number of expressions must not exceed the number of index-types in the array declaration. It is an error if the result of each expression is not assignment-compatible with the corresponding index-type.

In indexing a multi-dimensional array, you can use either multiple indexes or multiple expressions within an index. The two forms are equivalent. For example,

```
m[i][j]
```

is equivalent to

```
m[i, j]
```

A string value can be indexed by only one index expression, whose value must be in the range $1..n$, where n is the current length of the string value. The effect is to access one character of the string value, and the type of the character value accessed is *char*.

Any value assigned to a component-variable of an array or string must be assignment-compatible with the component-type.

Note: When a string value is manipulated by assigning values to individual character positions, the *length* of the string is not affected. It is an error to access a character position in a string variable with an index less than one or greater than the length of the string variable. For example, suppose that *strval* has been declared as follows:

```
strval: string[10];
```

and that the assignment:

```
strval := 'abcde';
```

has been performed. A reference to:

```
strval[0]
```

would be an error since the reference contains an index less than 1. Likewise, a reference to:

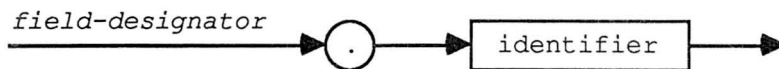
```
strval[6]
```

would be an error since the index is greater than the current length of the string. Note that these references would still be errors even if they occurred on the left-hand side of an assignment statement. To add a character or string to the end of another string, or to manipulate strings in other ways, use the standard procedures described in Section 10.5.

Note also that, when a program, procedure, or function block is entered, the length of a string variable is initially undefined. Therefore, the manipulation of string variables with indexes before string values have been assigned to these variables can lead to unpredictable results.

4.3.2 Records and Field-Designators

A specific field of a record variable is denoted by a variable-reference that refers to the record variable, followed by a field-designator that specifies a field of the record.



Examples of field-designators:

```
p2^.pregnant  
coord.theta
```

It is an error if the field-designator refers to a field in a variant that is not active (see Section 3.2.2).

Note that a field-designator need not be preceded by a variable-reference to its containing record in a statement within a **with** statement (see Section 6.2.4).

4.3.3 File-Buffers

Although a file-variable may have any number of components, only one component is accessible at any time. The position of the current component in the file is called the *current file position*. Program access to the current component is via a special variable associated with the file, called a *file-buffer*.

The file-buffer is implicitly declared when the file variable is declared. If f is a file variable with components of type t , then the associated file-buffer is a variable of type t .

The file-buffer associated with a file variable is denoted by a variable-reference that refers to the file variable, followed by the \wedge symbol. Thus, the file-buffer of file f is referenced by f^\wedge .

Chapter 9 describes standard procedures that are used to move the current file position within the file and to transfer data between the file-buffer and the current file component.

4.3.4 Pointers and Dynamic-Variables

The value of a pointer variable is either ***nil***, or a value that points to a dynamic-variable.

The dynamic-variable pointed to by a pointer variable is denoted by a variable-reference that refers to the pointer variable followed by the \wedge symbol.

Dynamic-variables and pointer values that point to them are created by the standard procedure *new* (see Section 10.1.1). Additionally, the *@* operator (see Section 5.1.6) and the standard procedure *pointer* (see Section 10.2.6) may be used to create pointer values that are not in fact pointers to dynamic-variables, but that will be treated as such.

The constant ***nil*** (see Section 3.4) does not point to any object. It is an error if you access a dynamic-variable when the pointer's value is undefined or is ***nil***.

Examples of references to dynamic-variables:

```
 $p1^\wedge$   
 $p1^\wedge.sibling^\wedge$ 
```

Note: Although the \wedge symbol is used to reference both file-buffers and dynamic-variables, this does not mean that a file variable can be treated like a pointer. Specifically, $ord(f)$, where f is a file variable reference, does not yield an address value (see Section 10.4.1). It is, in fact, an error.

Section 5

Expressions

5.1	Operators	47
5.1.1	Binary Operators: Order of Evaluation of Operands	47
5.1.2	Arithmetic Operators	47
5.1.3	Boolean Operators	49
5.1.4	Set Operators	50
5.1.5	Relational Operators	50
5.1.5.1	Comparing Ordinals	51
5.1.5.2	Comparing Reals	51
5.1.5.3	Comparing Strings	52
5.1.5.4	Comparing Packed-Strings	52
5.1.5.5	Comparing Sets	52
5.1.5.6	Comparing Pointers	52
5.1.5.7	Testing Set Membership	52
5.1.6	The @ Operator	53
5.1.6.1	The @ Operator with a Variable	53
5.1.6.2	The @ Operator with a Value Parameter	53
5.1.6.3	The @ Operator with a Variable Parameter	54
5.1.6.4	The @ Operator with a Procedure or Function Identifier	54
5.2	Function-Calls	54
5.3	Set-Constructors	55
5.4	Type-Casts	56

Section 5

Expressions

Expressions consist of *operators* and *operands*. Most operators in Pascal are *binary*, i.e. they take two operands; the rest are *unary* and take only one operand. The binary operators and their operands are denoted in the common algebraic fashion: the operands are given with the operator to act upon them in between, e.g., $a+b$. A unary operator is always immediately followed by its operand.

When more complex expressions are written, certain rules must be applied to determine which operands are associated with which operators. For instance, the expression:

$a+b*c$

can be interpreted as either $(a+b)*c$ or $a+(b*c)$. The *precedence rules* makes the interpretation unambiguous:

- When an operand appears between two operators of different precedence, it is bound to the operator with the higher precedence.
- When an operand is written between two operators of the same precedence, it is bound to the operator to the left.
- A parenthesized expression is always evaluated before it is applied as an operand.

Table 5-1 gives the precedence of the binary and unary operators:

Table 5-1
Precedence of
Operators

<i>Operators</i>	<i>Precedence</i>	<i>Categories</i>
@, not	highest	unary operators
*, /, div, mod, and	second	"multiplying" operators
+, -, or	third	"adding" operators & signs
=, <>, <, >, <=, >=, in	lowest	relational operators

Thus, $a+b*c$ is interpreted as $a+(b*c)$, since $*$ has a higher precedence than $+$, and $a+b-c$ is interpreted as $(a+b)-c$, since $+$ and $-$ have the same precedence.

Warning: Be aware that the range of any operator is not infinite. For instance, if a, b , and c are integer-type *integer* values, and if $a+b$ yields a value greater than *maxint*, then the evaluation $(a+b)-c$ will result in an error whereas $a+(b-c)$ may not. Whenever the order of evaluation of operators is critical, or is otherwise in doubt, parentheses may always be used to force a specific order.

Also be aware that, because the relational operators have the *lowest* precedence, an expression such as

$a < b \text{ or } c < d$

is erroneous. This is because the precedence rules want this to be interpreted as

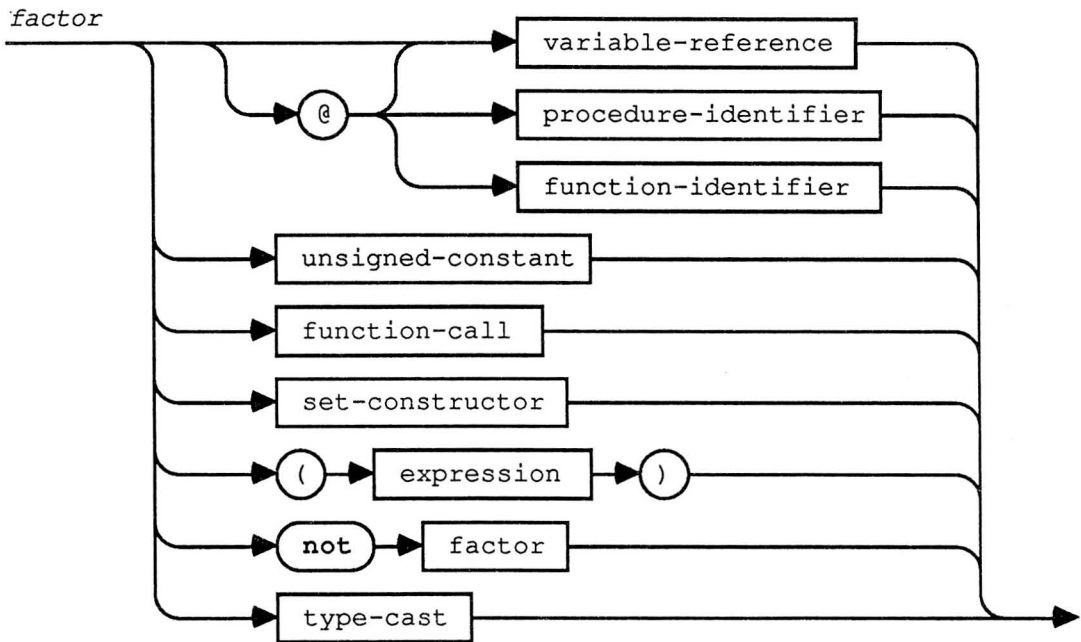
$a < (b \text{ or } c) < d$ '

which is not a valid expression (see below). Thus, such an expression must be written as

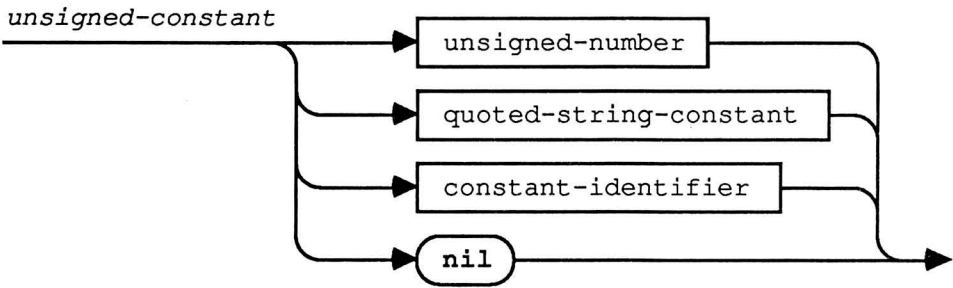
$(a < b) \text{ or } (c < d)$

The precedence rules are implicit from the syntax for expressions, which are built up from factors, terms, and simple-expressions.

The syntax for a *factor* allows the unary operators $@$ and *not* to be applied to a value:



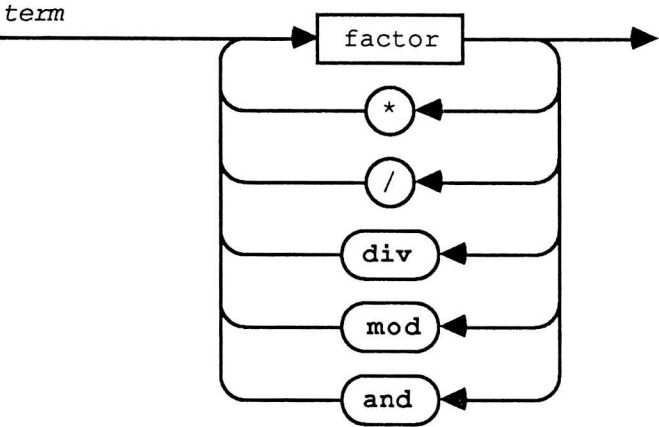
A *function-call* activates a function, and denotes the value returned by the function (see Section 5.2). A *set-constructor* denotes a value of a set-type (see Section 5.3). An *unsigned-constant* has the following syntax:



Examples of factors:

<i>x</i>	{variable-reference}
@ <i>x</i>	{pointer to a variable}
15	{unsigned-constant}
(<i>x</i> + <i>y</i> + <i>z</i>)	{sub-expression}
sin(<i>x</i> /2)	{function-call}
['A'..'F', 'a'..'f']	{set-constructor}
not <i>p</i>	{negation of a boolean}

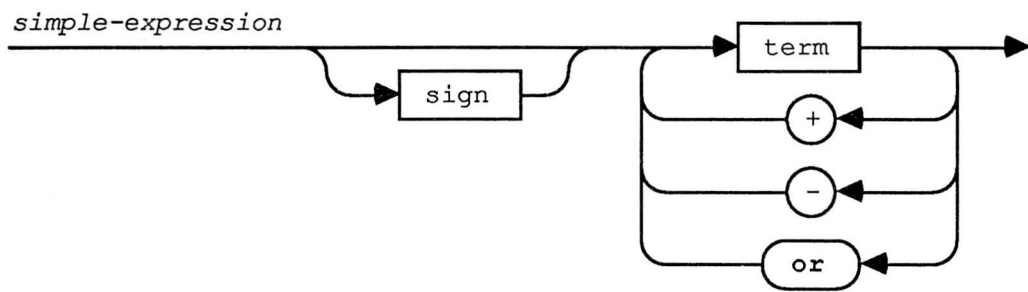
The syntax for a *term* allows the "multiplying" operators to be applied to factors:



Examples of terms:

<i>x</i> * <i>y</i>
<i>i</i> /(1- <i>i</i>)
<i>p</i> and <i>q</i>
(<i>x</i> <= <i>y</i>) and (<i>y</i> < <i>z</i>)

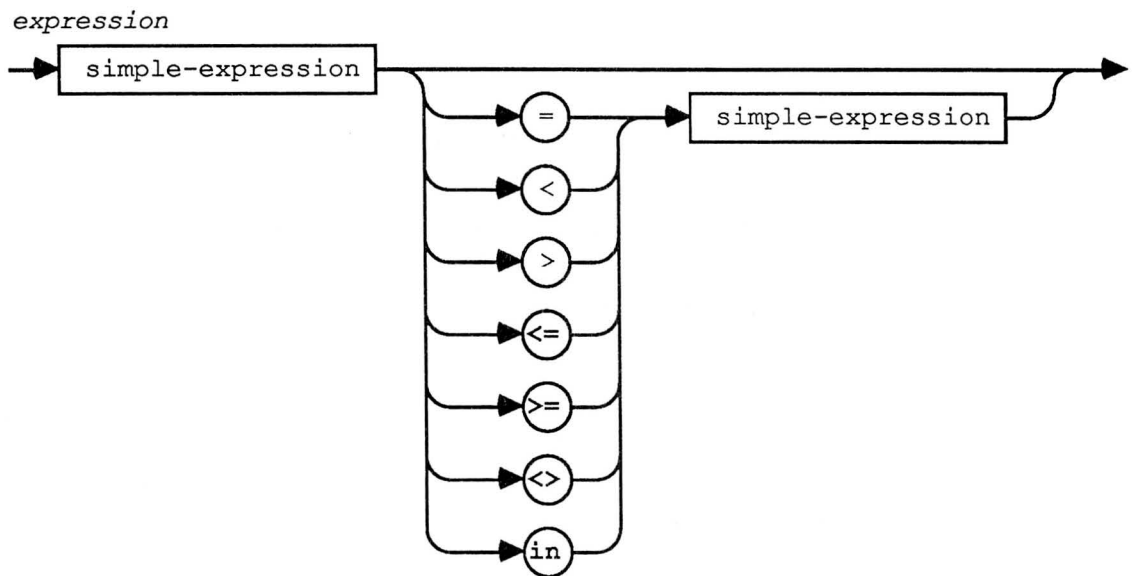
The syntax for a *simple-expression* allows the "adding" operators and signs to be applied to terms:



Examples of *simple-expressions*:

```
x+y
-x
hue1 + hue2
i*j + 1
```

The syntax for an *expression* allows the relational operators to be applied to simple-expressions:



Examples of *expressions*:

```
x = 1.5
p <= q
p = q and r
(i < j) = (j < k)
c in hue1
```

5.1 Operators

5.1.1 Binary Operators: Order of Evaluation of Operands

The order in which the operands of a binary operator are evaluated is unspecified.

Warning: A function, in the normal case, simply returns a value. However, a function may also alter the value of variables as a side-effect. Therefore if a function is one operand of a binary operator, and if it modifies the value of the other operand, then the evaluation of the operator may lead to unpredictable results. For instance, if a call to a function $f(x)$ modifies the value of x , then the evaluation of $x+f(x)$ may yield an unexpected result -- depending on whether or not $f(x)$ is evaluated before the x to the left of the $+$.

5.1.2 Arithmetic Operators

The types of operands and results for arithmetic binary and unary operations are shown in Tables 5-2 and 5-3 respectively.

Table 5-2
Binary Arithmetic
Operations

Operator	Operation	Operand Types	Type of Result
+	addition	<i>integer integer</i>	<i>integer</i>
-	subtraction	<i>integer longint longint longint</i>	<i>longint longint</i>
*	multiplication	<i>real-type</i>	<i>extended</i>
/	division	<i>integer-type or real-type</i>	<i>extended</i>
div	division with integer	<i>integer integer integer longint longint longint</i>	<i>integer longint longint</i>
mod	modulo	<i>integer integer integer longint longint longint</i>	<i>integer longint longint</i>
Note: The symbols +, -, and * are also used as set operators (See Section 5.1.4).			

Table 5-3
Unary Arithmetic Operations (Signs)

<i>Operator</i>	<i>Operation</i>	<i>Operand Types</i>	<i>Type of Result</i>
+	identity	<i>integer</i>	<i>integer</i>
-	sign-negation	<i>longint</i> <i>real-type</i>	<i>longint</i> <i>extended</i>

Any operand whose type is *subr*, where *subr* is a subrange of some ordinal host-type *ordtyp*, is treated as if it were of type *ordtyp*. Consequently an expression that consists of a single operand of type *subr* is itself of type *ordtyp*.

If both operands of the addition, subtraction, or multiplication operators are of the integer-type *integer*, the result is always of type *integer* and has a value determined by the normal mathematical rules for integer arithmetic. It is an error if the value of the result is outside the range *-maxint..maxint*.

If one or both operands of the addition, subtraction, or multiplication operators are of the integer-type *longint*, the result is always of type *longint* and has a value determined by the normal mathematical rules for integer arithmetic. It is an error if the value of the result is outside the range *-maxlongint..maxlongint*.

If one of the operands of the addition, subtraction, or multiplication operators is of a real-type, the result is always of type *extended* and has a value that is an approximation of the normal mathematical result. It is an error if the result is outside the range of values representable with the real-type *extended* (see Section 3.1.2).

Note: See Appendix D for more information on all arithmetic operations with operands or results of a real-type.

If the operand of the identity or sign-negation operator is of the integer-type *integer*, the result is always of type *integer* and the absolute value of the result is always identical to the absolute value of the operand.

If the operand of the identity or sign-negation operator is of a real-type, the result is always of type *extended* and the absolute value of the result is always identical to the absolute value of the operand.

If the operand of the identity or sign-negation operator is of the integer-type *longint*, the result is always of type *longint* and the absolute value of the result is always identical to the absolute value of the operand.

The value of *i/j* is always of type *extended* and has a value that is an approximation of the normal mathematical result. It is an error if the result is outside the range of values representable with the real-type *extended* (see Section 3.1.2). It is an error if *j=0*.

If the operands of the **div** operator are of the integer-type *integer*, the result is always of

type *integer*, and the value of *i div j* is the mathematical quotient of *i/j*, rounded toward zero. It is an error if *j=0*.

If one or both of the operands of the **div** operator are of the integer-type *longint*, the result is always of type *longint*, and the value of *i div j* is the mathematical quotients of *i/j*, rounded toward zero. It is an error if *j=0*.

In defining the result of the **mod** operator, assume there is a remainder operator, **rem**, that is defined as:

$$i \text{ rem } j = i - (i \text{ div } j) * j$$

Then

$$i \text{ mod } j = i \text{ rem } j, \text{ if } i \geq 0$$

and

$$i \text{ mod } j = j + i \text{ rem } j, \text{ if } i < 0 \quad \text{or zero}$$

Thus, the sign of the result of **mod** is always positive and the value of the result always agrees with the rules of modular arithmetic. It is an error if *j<=0*.

5.1.3 Boolean Operators

The types of operands and results for Boolean operations are shown in Table 5-4.

Table 5-4
Boolean Operations

Operator	Operation	Operand Types	Type of Result
or	disjunction	<i>boolean</i>	<i>boolean</i>
and	conjunction		
not	negation		

The result of a *boolean* operation is determined by the normal rules of boolean logic, e.g. *a and b* evaluates to *true* if and only if both *a* and *b* are true.

Warning: *Boolean* operators are somewhat unique in that the result of the operation may often be determined by the examination of only one operand. For example, the expression

$$(b < > 0) \text{ and } (a/b > 10)$$

is known to have the value *false* if *b=0* regardless of the value of *a*. However, the evaluation of the subexpression *(a/b>10)* may or

may not be performed in the evaluation of this expression. It is therefore best to assume that it will and avoid this kind of expression. On the other hand, if an operand of a boolean expression is a function with side-effects (i.e. it modifies a variable as well as returns a value), there is no guarantee that the function will be activated if the result of the operation can be determined solely by the evaluation of the other operand. It is therefore best to also avoid such expressions.

5.1.4 Set Operators

The types of operands and results for set operations are shown in Table 5-5.

Table 5-5
Set Operations

<i>Operator</i>	<i>Operation</i>	<i>Operand Types</i>	<i>Type of Result</i>
+	union	compatible set-types	(see below)
-	difference		
*	intersection		

The order of evaluation of member-groups and of expressions within member-groups is unspecified.

The results of the set operations are determined by the normal rules of set logic. i.e.

- An ordinal value c is in the set $a+b$ if and only if c is in a or in b .
- An ordinal value c is in the set $a-b$ if and only if c is in a and not in b .
- An ordinal value c is in the set $a*b$ if and only if c is in a and in b .

Given the result of a set operation, if the smallest ordinal value that is a member of that result is a and if the largest ordinal-value that is a member of the result is b , then the type of the result is **set of $a..b$** .

5.1.5 Relational Operators

The types of operands and results for relational operations are shown in Table 5-6.

Table 5-6
Relational Operations

Operator	Operation	Operand Types	Type of Result
=	equal	compatible simple, pointer, set, string, or packed-string types	boolean
<>	not equal		
<	less	compatible simple, string, or packed-string types	
>	greater		
<=	less/equal		
>=	greater/equal		
<=	subset of	compatible set-types	
>=	superset of		
in	member of	(see 5.1.5.7)	

5.1.5.1 Comparing Ordinals

When the operands of =, <>, <, >, >=, or <= are of an ordinal-type, they must be of compatible types. The result is the mathematical relation of their ordinalities.

5.1.5.2 Comparing Reals

When one operand of =, <>, <, >, >=, or <= is of a real-type, the other must be of a real-type or an integer-type. The result is the mathematical relation of the values represented as *extended* type values.

Note: Because real-type values are only approximations, the results of these operations may not always be as expected. For instance, if *aReal* is a variable of type *real* and *aDouble* is a variable of type *double*, and if the assignments

```
aReal := 1/3;
```

```
aDouble := 1/3;
```

have been performed, then the relation *aReal*=*aDouble* will return *false*. This is because the value of *aReal* is a representation of 1/3 to only 7-8 decimal digits and the value of *aDouble* is a representation of 1/3 to 15-16 decimal digits. Since the decimal (and even the binary) representation of 1/3 is a repeating sequence of digits, the 8 low-order decimal digits of *aDouble* will differ from the corresponding digits of *aReal* (when converted to *extended*), which will always be zero.

See Appendix D for more information on relational operations with operands of real-type.

5.1.5.3 Comparing Strings

When the relational operators `=`, `<>`, `<`, `>`, `>=`, and `>` are used to compare strings (see Section 3.3), they denote lexicographic ordering according to the ordering of the Macintosh character set (see Appendix J). Note that any two string values can be compared since all string values are compatible. Additionally, a *char* value is compatible with a string-type value, and when the two are compared, the *char* value is treated as a string-type value with length 1. When a packed-string-type value with *n* components is compared with a string-type value, it is treated as a string-type value with length *n*.

5.1.5.4 Comparing Packed-Strings

The relational operators `=`, `<>`, `<`, `>`, `<=`, and `>=` can also be used to compare two values of a packed-string-type if both have the same number of components. If that number of components is *n*, then the result is the same as if the values were string-type with each having a *length* of *n*.

5.1.5.5 Comparing Sets

If *a* and *b* are set operands, then

- *a*=*b* is *true* if and only if every member of *a* is a member of *b* and every member of *b* is a member of *a*; otherwise, *a*<>*b*.
- *a*<=*b* is *true* if and only if every member of *a* is also a member of *b*.
- *a*>=*b* is *true* if and only if every member of *b* is also a member of *a*.

Thus, *a*=*b* and *a*<>*b* denote the equivalence and non-equivalence of the sets *a* and *b* respectively, and *a*<=*b* and *a*>=*b* denote the inclusion of *a* in *b* and the inclusion of *b* in *a* respectively.

5.1.5.6 Comparing Pointers

The relational operators `=` and `<>` may be applied to compatible pointer-type operands. Two pointers are equal if and only if they point to the same object.

5.1.5.7 Testing Set Membership

The *in* operator yields the value *true* if the value of the ordinal-type operand is a member of the set-type operand; otherwise it yields the value *false*. The type of the left operand must be compatible with the base-type of the right operand.

5.1.6 The @ Operator

A pointer value that points to a variable, procedure, or function can be created with the @ operator. The operand and result types are shown in Table 5-7.

@ is a unary operator taking a single variable-reference or a procedure or function identifier as its operand and computing the value of its pointer. The type of the value is equivalent to the type of *nil*, i.e. it can be assigned to any pointer variable.

Table 5-7
Pointer Operation

<i>Operator</i>	<i>Operation</i>	<i>Operand</i>	<i>Type of Result</i>
@	pointer formation	variable-reference or procedure or function identifier	same as <i>nil</i>

Warning: The @ operator is not a standard feature of Pascal, and its indiscriminate use is not recommended. It is intended to be used in conjunction with the Macintosh Toolbox routines (see Appendices C and E).

5.1.6.1 The @ Operator with a Variable

For an ordinary variable (not a parameter), the use of @ is straightforward. For example, if we have the declarations

```
type
  twochar = packed array[0..1] of char;
var
  int: integer;
  twocharptr: ^twochar;
```

then the statement

```
twocharptr := @int
```

causes *twocharptr* to point to *int*. Now *twocharptr*[^] is a reinterpretation of the bit value of *int* as though it were a **packed array**[0..1] of *char*.

The operand of @ cannot be applied to a component of a **packed** variable.

5.1.6.2 The @ Operator with a Value Parameter

When @ is applied to a formal value parameter, the result is a pointer to the location containing the actual value, which is on a runtime stack. Suppose that *foo* is a formal value parameter in a procedure and *fooPtr* is a pointer variable. If the procedure executes the statement

```
fooptr := @foo
```

then $fooptr^{\wedge}$ is a reference to the value of *foo*. Note that if the actual-parameter is a variable-reference, $fooptr^{\wedge}$ is not a reference to the variable itself; it is a reference to the value taken from the variable and stored on the stack.

5.1.6.3 The @ Operator with a Variable Parameter

When @ is applied to a formal variable parameter, the result is a pointer to the actual-parameter. Suppose that *fum* is a formal variable parameter of a procedure, *fie* is a variable passed to the procedure as the actual-parameter for *fum*, and *fumptr* is a pointer variable.

If the procedure executes the statement

```
fumptr := @fum
```

then *fumptr* is a pointer to *fie* and $fumptr^{\wedge}$ is a reference to *fie* itself.

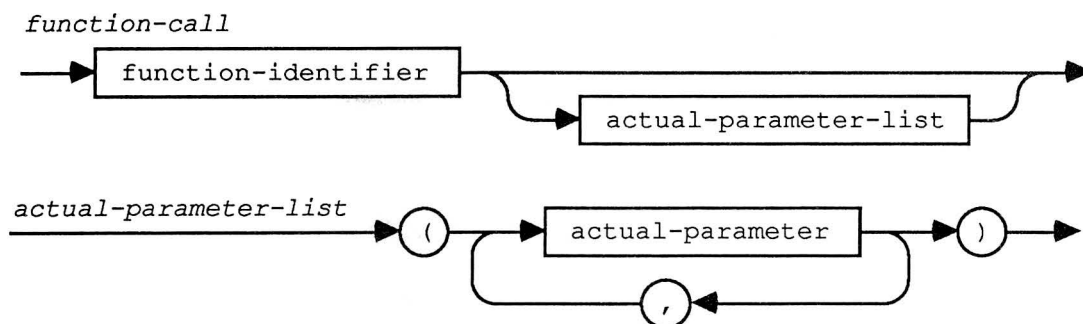
5.1.6.4 The @ Operator with a Procedure or Function Identifier

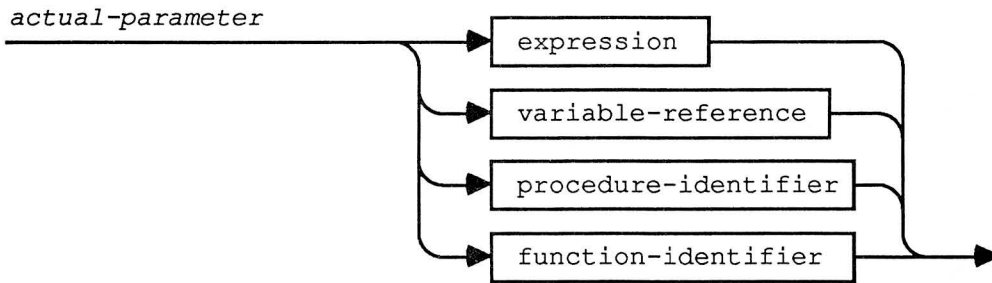
It is possible to apply @ to a procedure or a function, yielding a pointer to the procedure's or function's *entry-point*. This pointer actually points to a *jump table entry* for the routine; see Chapter 11 of the *User's Guide* for more information.

Lightspeed Pascal provides no mechanism for using such a pointer. The typical use for a procedure pointer is to pass it to a Macintosh Toolbox routine. The @ operator can not be applied to predefined, inline, Toolbox, or nested routines.

5.2 Function-Calls

A function-call specifies the activation of the function denoted by the function-identifier. The result returned by the function activation is subsequently used as an expression value. If the corresponding function-declaration contains a list of formal-parameters, then the function-call must contain a corresponding list of actual-parameters. Each actual-parameter is substituted for the corresponding formal-parameter as described in Section 7.3.





A function-identifier is any identifier that has been declared to denote a function.

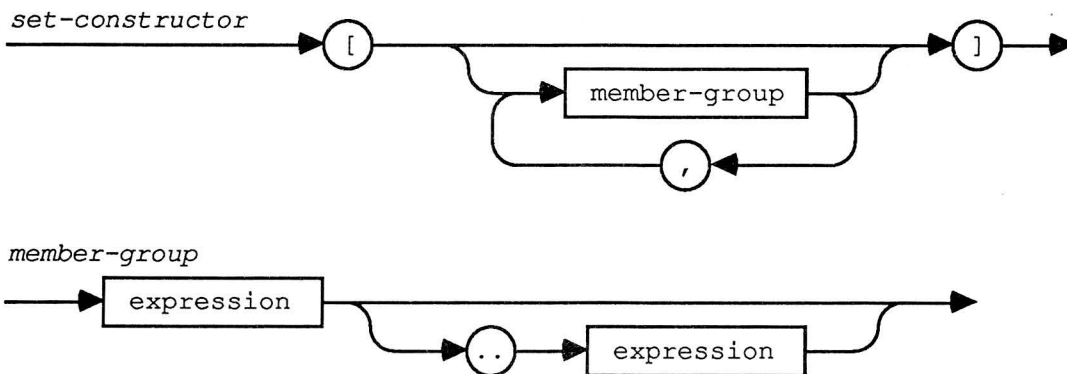
Examples of function-calls:

```

sum(a, 63)
gcd(147, k)
sin(x+y)
eof(f)
ord(f^)
```

5.3 Set-Constructors

A set-constructor denotes a value of a set-type, and is formed by writing expressions within [brackets]. Each expression denotes a value of the set.



The notation `[]` denotes the empty set, which is assignment-compatible to every set-type. Any member-group `x . . y` denotes as set members all values in the range `x . . y`. If the value of `x` is greater than the value of `y`, then `x . . y` denotes no members and `[x . . y]` denotes the empty set.

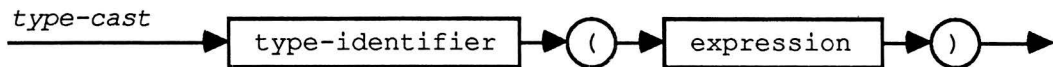
All expression values in the member-groups of a particular set-constructor must be of compatible ordinal-types. If `a` is the smallest ordinal-value in the resulting set, and if `b` is the largest ordinal-value in the resulting set, then the base-type of the resulting set is `a . . b`.

Examples of set-constructors:

```
[red, c, green]
[1, 5, 10..k mod 12, 23]
['A'..'Z', 'a'..'z', chr(xcode)]
```

5.4 Type-Casts

Type casting provides a way to change the type of an expression to another type.



For ordinal and pointer types, the result of this operation is an expression of type *type-identifier* whose (ordinal) value is obtained by converting the original expression. This conversion may involve truncation or extension of the original value if the storage size of the expression is changed.

For non-ordinal types, the result of this operation is an expression of type *type-identifier* whose internal representation (i.e., the pattern of bits which comprise its value) is the same as the internal representation of the original expression. In particular, the storage size of a (non-ordinal) expression may not be changed by a type cast.

Examples of type-casts:

```
boolean (1)          ptr (longint (p)+1)      ptr (-1)
longint (@proc)      color (x)
```

Section 6

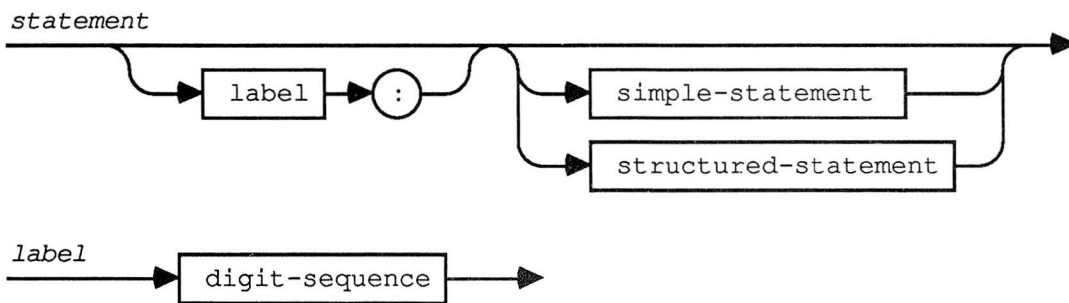
Statements

6.1	Simple-Statements	59
6.1.1	Assignment-Statements	59
6.1.2	Procedure-Statements	60
6.1.3	Goto-Statements	61
6.2	Structured-Statements	63
6.2.1	Compound-Statements	63
6.2.2	Conditional-Statements	63
6.2.2.1	If-Statements	64
6.2.2.2	Case-Statements	65
6.2.3	Repetitive-Statements	66
6.2.3.1	Repeat-Statements	66
6.2.3.2	While-Statements	67
6.2.3.3	For-Statements	67
6.2.4	With-Statements	70

Section 6

Statements

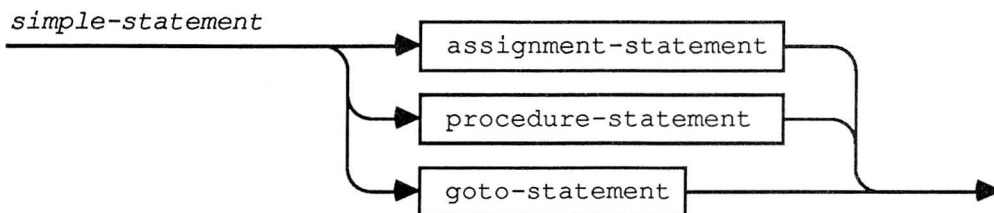
Statements denote algorithmic actions, and are executable. They can be prefixed by labels and a labeled statement can be referenced by a goto-statement.



A digit-sequence used as a label must be in the range 0..9999, and must first be declared as described in Section 2.1.

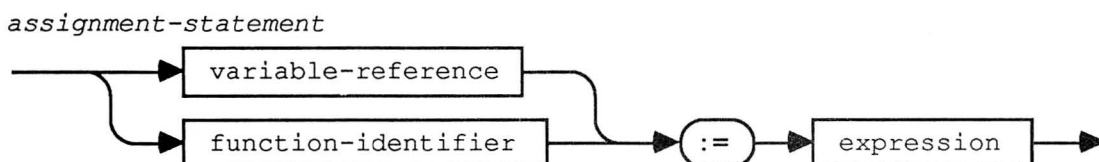
6.1 Simple-Statements

A simple-statement is a statement that does not contain any other statement.



6.1.1 Assignment-Statements

The syntax for an assignment-statement is as follows:



The assignment-statement can be used in two ways:

- To replace the current value of a variable by a new value as specified by an expression
- To specify an expression whose value is to be returned by a function.

The expression must be assignment-compatible with the type of the variable or the result-type of the function as described in Section 3.5.3.

Note: It is not specified whether the variable-reference is evaluated before or after the evaluation of the expression. However, once the reference is established, it is not altered by the remaining execution of the assignment-statement. Thus, the outcome of

$$a[x] := f(x)$$

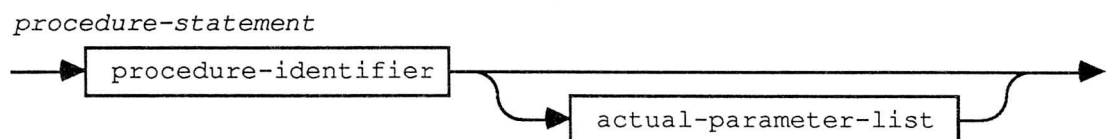
depends on whether f modifies x and, if so, whether $f(x)$ is evaluated before or after $a[x]$.

Examples of assignment-statements:

```
x := y+z;  
p := (1<=i) and (i<100);  
i := sqr(k) - (i*j);  
hue1 := [blue,succ(c)]
```

6.1.2 Procedure-Statements

A procedure-statement specifies the activation of the procedure denoted by the procedure-identifier. If the corresponding procedure-declaration contains a list of formal-parameters, then the procedure-statement must contain a corresponding list of actual-parameters. Each actual-parameter is substituted for the corresponding formal-parameter as described in Section 7.3.



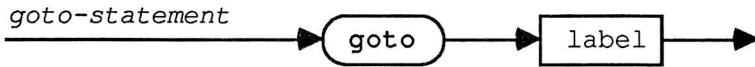
Note: The order in which actual parameters are evaluated and bound to their formal parameters is unspecified.

Examples of procedure-statements:

```
PrintHeading;  
Transpose(a,n,m);  
Bisect(fct,-1.0,+1.0,x)
```

6.1.3 Goto-Statements

A goto-statement causes the statement prefixed by the label that is referenced in the goto-statement to be the next statement executed.



Note: The constants that introduce cases within a case-statement (see Section 6.2.2.2) are not labels, and cannot be referenced in goto-statements.

A goto-statement G can **goto** a labeled statement S if and only if one of the following is true:

- S is a statement that *contains* G. For example:

```
1: if ... then
    2: begin
        ...
        3: begin
            ...
            goto {either 1, 2 or 3 is permissible,
                  4 is not}
            ...
            end
        ...
    end
else
    4: begin
        ...
        end
```

- S is a statement of a statement-list that contains G. For example:

```

begin
  ...
1: ...;
  ...
  begin
    ...
    2: ...;
    ...
    begin
      ...
      goto {either 1, 2, 3, or 4 is permissible, 5 is not}
    ...
    end
  ...
  3: ...;
  ...
  end
  ...
  4: ...;
  ...
end;
begin
  5: ...
end

```

- S is a statement in a block that contains the block containing G, provided that S is a statement of the outermost statement-list of its block. For example:

```

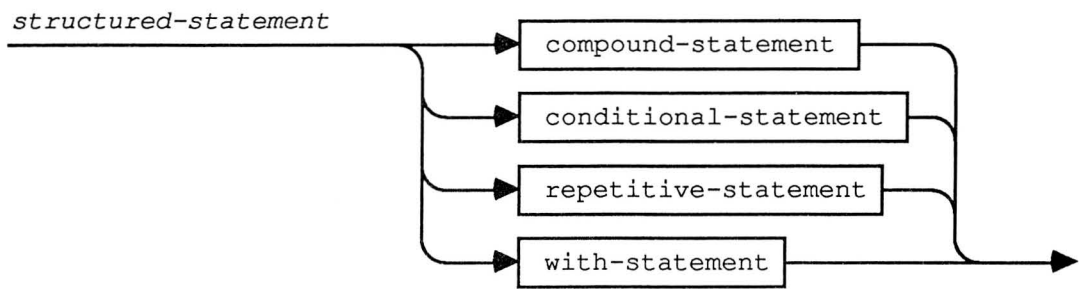
program a(...);
  procedure b;
    procedure c;
      begin
        goto {either 2 or 4 is permissible; 1 and 3 are not}
      end;
    procedure d;
      begin
        1: ...
      end;
    begin
      2: ...;
      begin
        3: ...
      end
    end;
  begin
    4: ...
  end.

```

When the destination of a **goto** is in a block *b* that does not contain the **goto**, every block activation (see Section 2.3) that has occurred since the most recent activation of *b* is terminated.

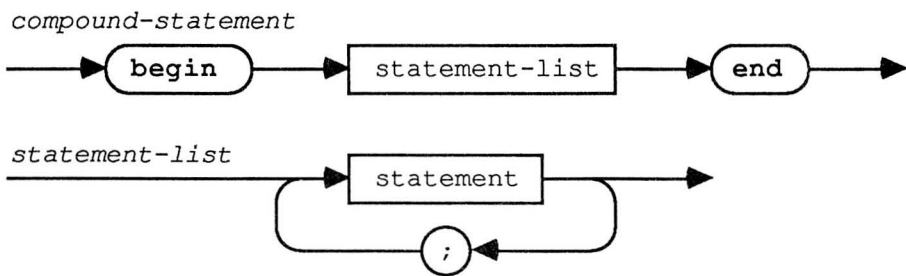
6.2 Structured-Statements

Structured-statements are made up of of other statements that are to be executed either conditionally (conditional-statements), repeatedly (repetitive-statements), or in sequence (compound-statement or with-statement).



6.2.1 Compound-Statements

The compound-statement specifies that its component statements are to be executed in the same sequence as they are written. The semicolon is a statement separator, not terminator.



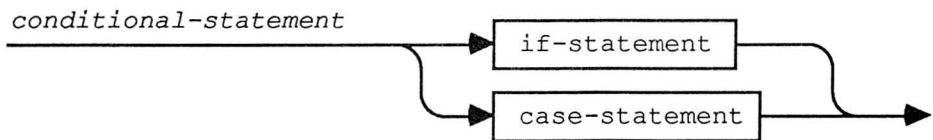
Example of compound-statement:

```
begin
  z := x;
  x := y;
  y := z
end
```

An important use of the compound-statement is to group more than one statement into a single statement in contexts where the Pascal syntax only allows one statement. The symbols *begin* and *end* act as "statement brackets." Examples of this will be seen in Section 6.2.3.2.

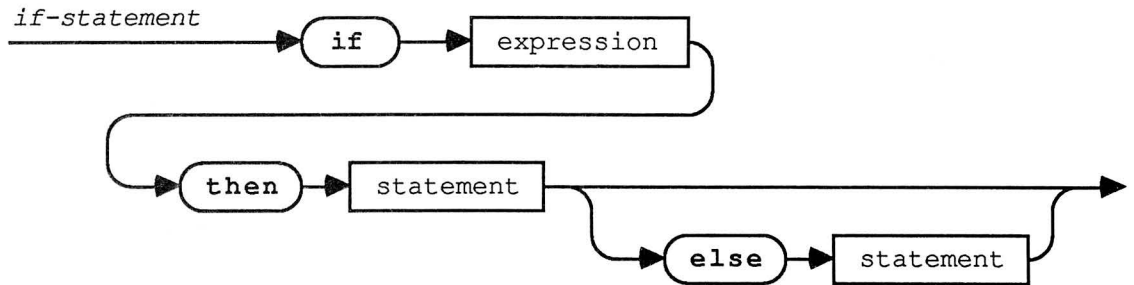
6.2.2 Conditional-Statements

A conditional-statement selects for execution a single one (or none) of its component statements.



6.2.2.1 If-Statements

The syntax for if-statements is as follows:



The expression must yield a result of the standard type *boolean*. If the expression yields the value *true*, then the statement following the **then** is executed.

If the expression yields *false*, and the **else** part is present, the statement following the **else** is executed; if the **else** part is not present, then execution proceeds with the next statement following the **if** statement.

The syntactic ambiguity arising from something like:

```
if e1 then if e2 then s1 else s2
```

is resolved by interpreting it as being equivalent to:

```
if e1 then  
  begin  
    if e2 then  
      s1  
    else  
      s2  
  end
```

rather than:

```
if e1 then  
  begin  
    if e2 then  
      s1  
    end  
  else  
    s2
```

In other words, an **else** is always associated with the closest **if** that is not already associated with an **else**.

Examples of if-statements:

```

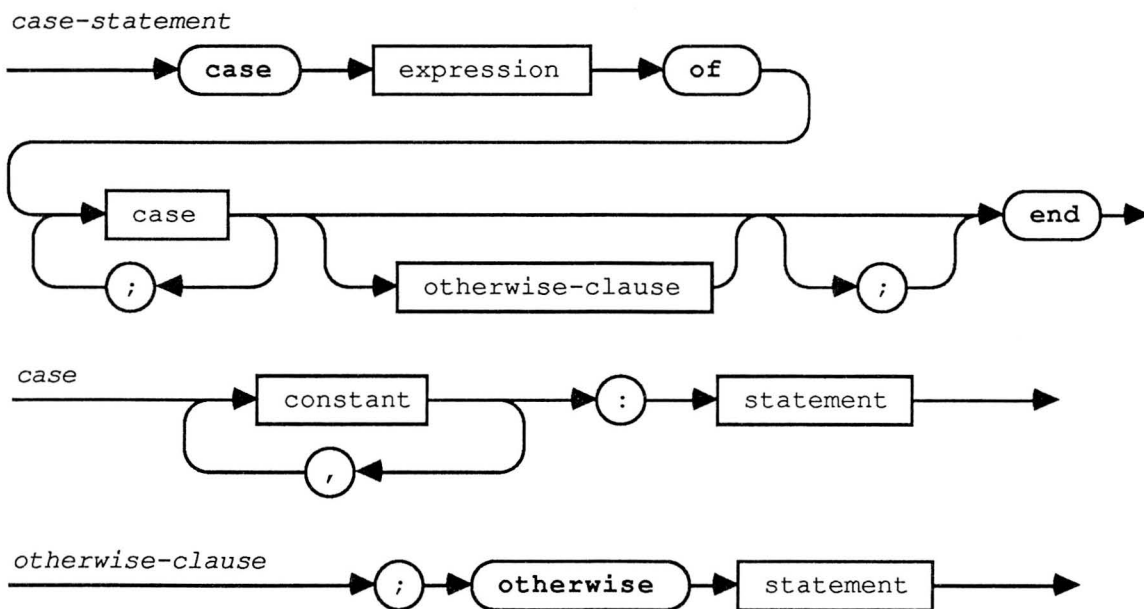
if x < 1.5 then
  z := x+y
else
  z := 1.5

if p1 <> nil then
  p1 := p1^.father

```

6.2.2.2 Case-Statements

The case-statement contains an expression (the *selector*) and a list of statements. Each statement must be prefixed with one or more constants (called *case-constants*), or with the reserved word **otherwise**. All the case-constants must be distinct and must be of an ordinal-type that is compatible with the type of the selector. A case-constant can be of type *longint*.



The case-statement specifies execution of the statement prefixed by a case-constant equal to the current value of the selector. If no such case-constant exists and an **otherwise** part is present, the statement following the word **otherwise** is executed; if no **otherwise** part is present, it is an error.

Examples of case-statements:

```

case operator of
  plus:  x := x+y;
  minus: x := x-y;
  times: x := x*y
end

```

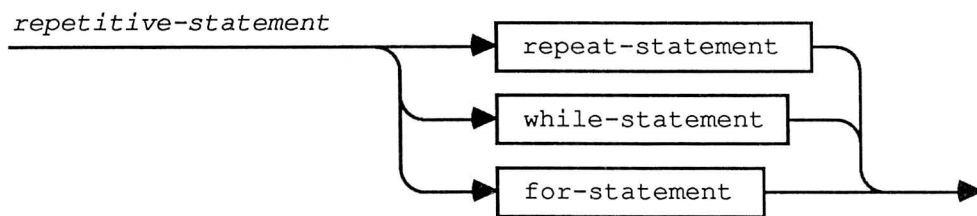
```

case i of
  1: x      := sin(x);
  2: x      := cos(x);
  3,4,5: x := exp(x);
otherwise
  x := ln(x)
end

```

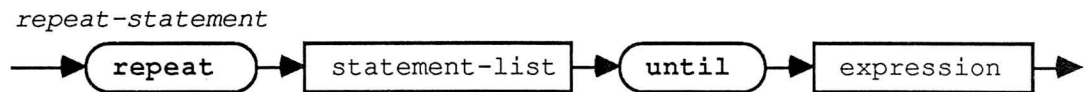
6.2.3 Repetitive-Statements

Repetitive-statements specify that certain statements are to be executed repeatedly.



6.2.3.1 Repeat-Statements

A repeat-statement contains an expression that controls the repeated execution of a sequence of statements contained within the repeat-statement.



The expression must yield a result of the standard type *boolean*. The statements between the symbols *repeat* and *until* are repeatedly executed in sequence until, at the end of a sequence, the expression yields the value *true*. The sequence of statements is executed at least once, because the expression is evaluated after the execution of each sequence.

Examples of repeat-statements:

```

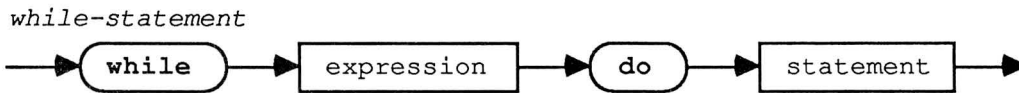
repeat
  k := i mod j;
  i := j;
  j := k
until j = 0

repeat
  process(f^);
  get(f)
until eof(f)

```

6.2.3.2 While-Statements

A while-statement contains an expression that controls the repeated execution of a statement (which may be a compound-statement).



The expression must yield a result of the standard type *boolean*. It is evaluated before the contained statement is executed. The contained statement is repeatedly executed as long as the expression yields the value *true*. If the expression yields *false* at the beginning, the statement is not executed.

The while-statement:

```
while b do  
    body
```

is equivalent to:

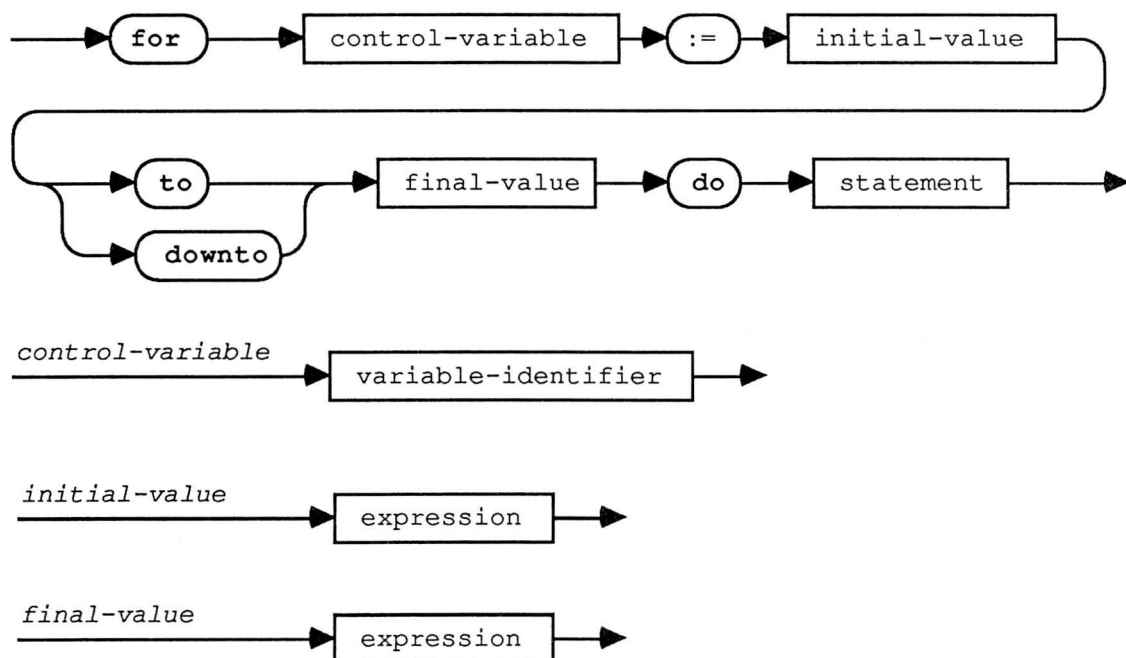
```
if b then  
    repeat  
        body  
    until not b
```

Examples of while-statements:

```
while a[i] <> x do  
    i := i+1  
  
while i>0 do  
    begin  
        if odd(i) then  
            z := z*x;  
        i := i div 2;  
        x := sqr(x)  
    end  
  
while not eof(f) do  
    begin  
        process(f^);  
        get(f)  
    end
```

6.2.3.3 For-Statements

The for-statement causes a statement (possibly a compound-statement) to be repeatedly executed while a sequence of values is assigned to a variable called the *control-variable*.



The control-variable must be a variable-identifier (without any qualifier) denoting a variable that is declared to be local to the block containing the for-statement. The control-variable must be of an ordinal-type, and the initial and final values must be of a type assignment-compatible with this type.

On entering a for-statement, the initial-value and the final-value are determined once (and only once) for the remainder of the execution of the for-statement.

Loosely speaking, the statement contained by the for-statement is executed once for every value in the range *initial-value*.*final-value*. With a for-statement using **to**, the *control-variable* has the value *initial-value* the first time, *succ(initial-value)* the second time, and so on. If the *initial-value* is greater than the *final-value*, then the contained statement is not executed. With a for-statement using **downto**, the *control-variable* has the value *initial-value* the first time, *pred(initial-value)* the second time, and so on. If the *initial-value* is less than the *final-value*, then the contained statement is not executed.

It is an error if the value of the control-variable is altered by execution of the contained statement. After a for-statement is executed, the value of the control-variable is undefined, unless the execution of the for-statement was terminated by a **goto** out of the for-statement.

Apart from these restrictions, the for-statement:

```
for v := e1 to e2 do
  body
```

is equivalent to:

```
begin
  temp1 := e1;
  temp2 := e2;
  if temp1 <= temp2 then
    begin
      v := temp1;
      body;
      while v <> temp2 do
        begin
          v := succ(v);
          body
        end
      end
    end
  end
```

and the for-statement:

```
for v := e1 downto e2 do
  body
```

is equivalent to:

```
begin
  temp1 := e1;
  temp2 := e2;
  if temp1 >= temp2 then
    begin
      v := temp1;
      body;
      while v <> temp2 do
        begin
          v := pred(v);
          body
        end
      end
    end
  end
```

where *temp1* and *temp2* are auxiliary variables of the host-type of the variable *v* that do not occur elsewhere in the program; they are used to resolve the expressions *e1* and *e2* upon entering the statement's body.

Examples of for-statements:

```
for i := 2 to 63 do
  if a[i] > max then
    max := a[i]

for i := 1 to n do
  for j := 1 to n do
```

```

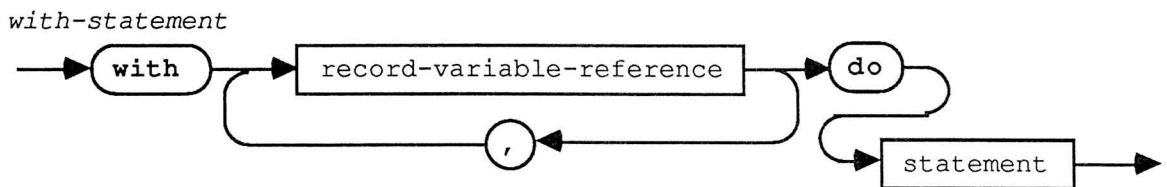
    begin
        x := 0;
        for k := 1 to n do
            x := x + m1[i,k]*m2[k,j];
        m[i,j] := x
    end

    for c := red to blue do
        q(c)

```

6.2.4 With-Statements

The syntax for a with-statement is



The occurrence of a record-variable-reference in a with-statement affects the way the compiler processes variable-references within the statement following the word **do**. Within a with-statement, fields of the record-variable can be referenced directly by their field-identifiers, without making explicit reference to the record-variable.

Example of a with-statement:

```

with date do
    if month = 12 then
        begin
            month := 1;
            year := year + 1
        end
    else
        month := month + 1

```

This is equivalent to:

```

if date.month = 12 then
    begin
        date.month := 1;
        date.year := date.year + 1
    end
else
    date.month := date.month + 1

```

Within a with-statement, each variable-reference is checked to see if it can be interpreted as a field of the record. If so, it is always interpreted as such, even if a variable with the same name is accessible also. For instance, suppose that we have the following declarations:

```

type
  recTyp = record
    foo: integer;
    bar: real;
  end;
var
  bar: recTyp;
  foo: integer;

```

The identifier *foo* can refer both to a field of the record variable *bar* and to a variable of type *integer*. In the statement

```

with bar do
  begin
    ...
    foo := 36;
    bar := 2.5;
    ...
  end

```

the *bar* between the **with** and the **do** is a reference to the variable *bar*, but *foo* is a reference to the field *bar.foo*, not the variable *foo*. Likewise, the reference to *bar* within this with-statement refers to *bar.bar*.

The statement:

```

with  $v_1, v_2, \dots, v_n$  do
  S

```

is equivalent to:

```

with  $v_1$  do
  with  $v_2$  do
    ....
    with  $v_n$  do
      S

```

Thus, if v_n in the above statements is a field of both v_1 and v_2 , it is interpreted to mean $v_2.v_n$ and not $v_1.v_n$.

If the selection of a variable in the record-variable-list involves the indexing of an array or the de-referencing of a pointer, these actions are executed only once before the component statement is executed.

Section 7

Procedures and Functions

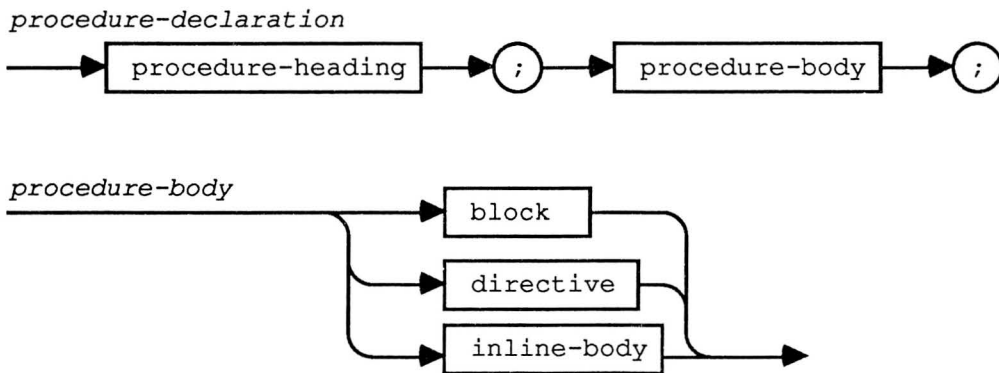
7.1	Procedure-Declarations	75
7.1.1	Forward-Declarations	76
7.1.2	External-Declarations	77
7.1.3	Inline-Declarations	77
7.2	Function-Declarations	78
7.3	Parameters	80
7.3.1	Value Parameters	81
7.3.2	Variable Parameters	81
7.3.3	Procedural Parameters	82
7.3.4	Functional Parameters	85
7.3.5	Parameter List Compatibility	85

Section 7

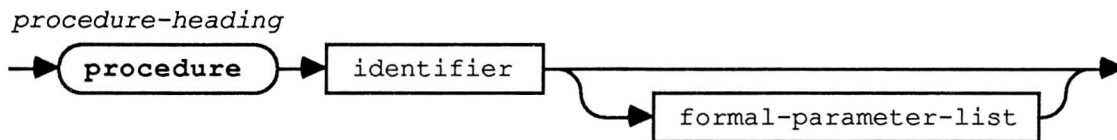
Procedures and Functions

7.1 Procedure-Declarations

A procedure-declaration associates an identifier with a block as a procedure so that it can be activated by a procedure-statement (see Section 6.1.2).



The procedure-heading specifies the identifier for the procedure, and the formal parameters (if any).



The syntax for a formal-parameter-list is given in Section 7.3.

A procedure is activated by a procedure-statement (see Section 6.1.2), which gives the procedure's identifier and any actual-parameters required by the procedure. The statements to be executed upon activation of the procedure are specified by the statement-part of the procedure's block. If the procedure's identifier is used in a procedure-statement within the procedure's block, the procedure is executed *recursively* (see Section 2.3).

Example of a procedure-declaration:

```
procedure ReadInteger( var f : text; var x : integer );  
    var  
        value, digit : integer;  
begin  
    while (f^ = ' ') and not eof(f) do  
        get(f);  
    value := 0;  
    while (f^ in ['0'..'9']) and not eof(f) do  
        begin  
            digit := ord(f^) - ord('0');  
            value := 10*value + digit;  
            get(f)  
        end;  
    x := value  
end;
```

7.1.1 Forward-Declarations

A procedure-declaration that has the directive *forward* instead of a block is called a *forward declaration*. Somewhere after the forward declaration, the procedure is actually defined by a *defining-declaration* -- a procedure-declaration that uses the same procedure-identifier, but omits the formal-parameter-list, and includes a block. The forward declaration and the defining-declaration must be in the same procedure-and-function-declaration-part, but need not be contiguous; that is, other procedures or functions can be declared between them and can call the procedure that has been declared forward. This permits *mutual recursion*.

The forward declaration and the defining-declaration constitute a complete declaration of the procedure. The procedure is considered to be declared at the place of the forward declaration.

Example of a forward declaration:

```
procedure walter( m,n : integer);  
forward;  
  
procedure clara( x, y : real);  
begin  
    ...  
    walter( 4, 5 );  
    ...  
end;  
  
procedure walter;  
begin  
    ...  
    clara( 8.3, 2.4 );  
    ...  
end;
```

7.1.2 External-Declarations

A procedure-declaration that has the directive *external* instead of a block is called an *external declaration*. External procedures and functions are used to declare the Pascal interface to a separately compiled or assembled routine. The external code must be linked with the rest of the program before execution.

Example of an external-declaration:

```
procedure DoInits(num:integer);  
external;
```

This means that *DoInits* is an external procedure that will be linked with the rest of the program before execution.

Note: It is the programmer's responsibility to insure that the external procedure is compatible with the external declaration in the Pascal program.

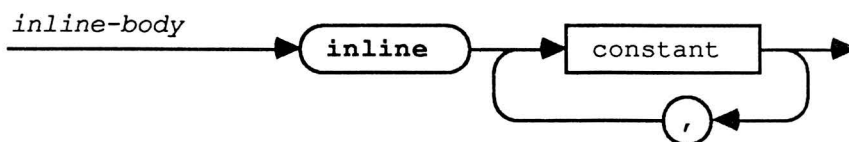
A unit (see Section 8.3) may declare a procedure in the interface-part and then implement this procedure by an external declaration. For example,

```
unit a;  
interface  
  procedure foo (arg:integer);  
implementation  
  procedure foo;  
    external;  
end.
```

This description of external procedures also applies to external functions.

7.1.3 Inline-Declarations

A procedure-declaration that has the word-symbol *inline* followed by one or more integer constants instead of a block is called an *inline declaration*. Inline procedures and functions are used to embed machine code in a Pascal program.



When a procedure is normally called, code is generated that reserves one or two words of function result (if a function is being called), and pushes the procedure's arguments (if any). Then a JSR instruction is generated. By declaring a routine as *inline*, the compiler will cause the constants that follow the word-symbol *inline* to be generated in place of the JSR instruction. Each constant represents one word (16 bits) and is generated in the order given.

Example of an inline-declaration:

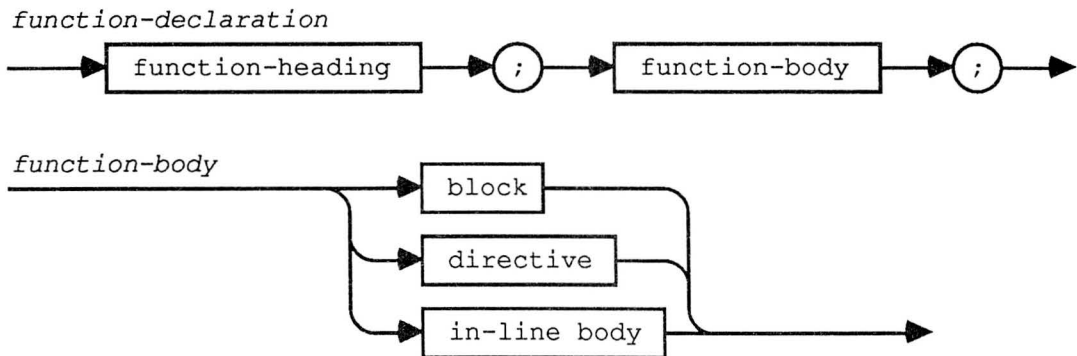
```
procedure trap (tos : longint);  
inline $A9ED;
```

- Notes:**
- 1) It is the programmer's responsibility to observe the proper rules for adjusting the stack, saving registers, etc.
 - 2) An inline procedure declared in a unit's interface section has no corresponding declaration in the implementation section.
 - 3) A forward declaration or interface procedure declaration may not be later defined as an inline declaration.

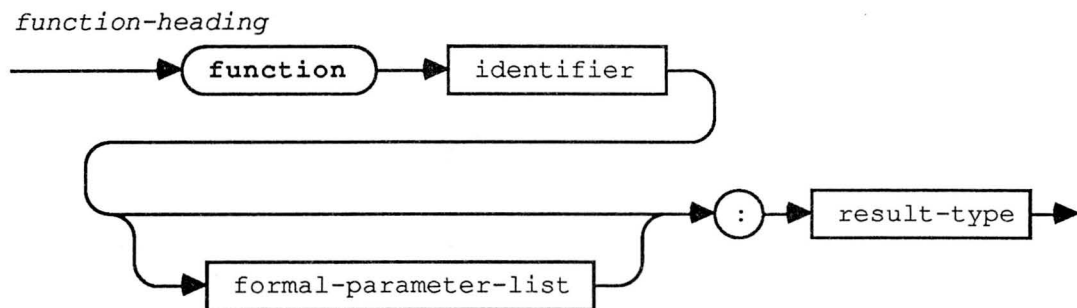
This description of inline procedures also applies to inline functions.

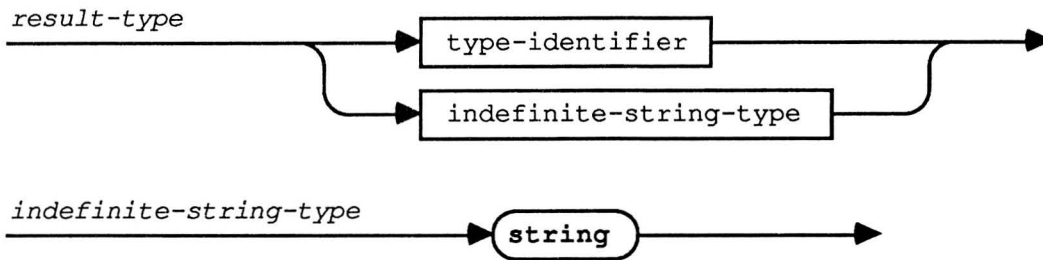
7.2 Function-Declarations

A function-declaration serves to declare a part of the program that computes and returns a value of some type.



The function-heading specifies the identifier for the function, the formal parameters (if any), and the type of the function result.





The syntax for a formal-parameter-list is given in Section 7.3.

A function is activated by the evaluation of a function-call (see Section 5.2), which gives the function's identifier and any actual-parameters required by the function. The function-call appears as an operand in an expression. The expression is evaluated by executing the function and, in effect, replacing the function-call with the value returned by the function.

The statements to be executed upon activation of the function are specified by the statement-part of the function's block. This block should normally contain at least one assignment-statement (see Section 6.1.1) that assigns a value to the function-identifier. The result of the function is the last value assigned. If no such assignment-statement exists, or if it exists but is not executed, the value returned by the function is undefined, which is an error.

If the function's identifier appears as an operand in an expression within the function's block, the function is executed *recursively*.

Examples of function-declarations:

```
function max( a : vector; n : integer ) : real;
  var
    x : real;
    i : integer;
begin
  x := a[1];
  for i := 2 to n do
    if x < a[i] then
      x := a[i];
  max := x
end;
```



```

function power( x : real; y : integer ) : real;
  var
    w, z : real;
    i     : integer;
begin
  w := x;
  z := 1;
  i := y;
  while i > 0 do
    begin
      { z*(w**i) = x ** y }
      if odd(i) then
        z := z*w;
        i := i div 2;
        w := sqr(w)
      end;
      { z = x**y }
      power := z
    end;

```

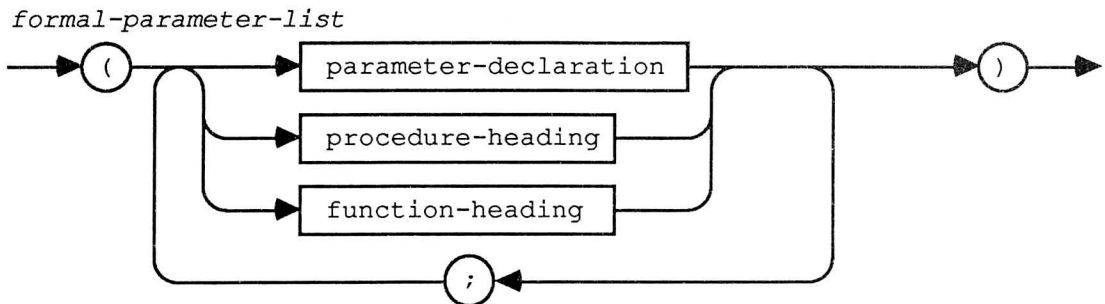
A function can be declared forward in the same manner as a procedure (see Section 7.1 above). This permits *mutual recursion*.

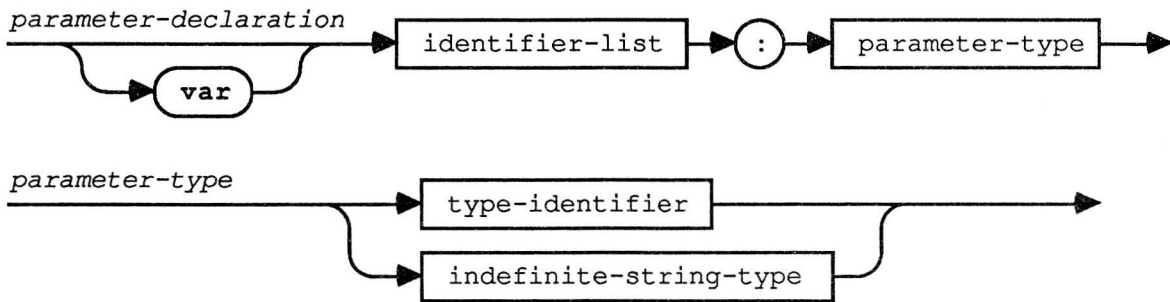
7.3 Parameters

A formal-parameter-list may be part of a procedure-declaration or function-declaration, or it may be part of the declaration of a procedural or functional parameter.

If it is part of a procedure-declaration or function-declaration, it declares the formal parameters of the procedure or function. Each parameter so declared is local to the procedure or function being declared, and can be referenced by its identifier in the block associated with the procedure or function.

If it is part of the declaration of a procedural or functional parameter, it declares the formal parameters of the procedural or functional parameter. In this case there is no associated block and the identifiers of parameters in the formal-parameter-list are not significant (see Sections 7.3.3 and 7.3.4 below).





There are four kinds of parameters: *value parameters*, *variable parameters*, *procedural parameters*, and *functional parameters*. They are distinguished as follows:

- A parameter-group preceded by **var** is a list of variable parameters.
- A parameter-group without a preceding **var** is a list of value parameters.
- A procedure-heading or function-heading denotes a procedural or functional parameter; see Sections 7.3.3 and 7.3.4 below.

Note: The type of a formal-parameter is denoted by either a type-identifier or the word-symbol **string**. Thus, to use a type such as **array**[0..255] **of** *char* as the type of a parameter, you must declare a type-identifier for this type:

```

type
    chararray = array[0..255] of char;
  
```

The identifier *chararray* can then be used in a formal-parameter-list to denote the type.

The interpretation of **string** as a parameter-type is described below.

7.3.1 Value Parameters

For a value parameter, the corresponding actual-parameter in a procedure-statement or function-call (see Sections 5.2 and 6.1.2) must be an expression, and its value must not be of file-type or of any structured-type that contains a file-type. The formal value parameter denotes a variable local to the procedure or function. The current value of the expression is assigned to the formal value parameter upon activation of the procedure or function. The actual-parameter must be assignment-compatible with the type of the formal value parameter (see Section 3.5.3). If the parameter-type is **string**, then the formal parameter is given a size attribute of 255.

7.3.2 Variable Parameters

For a variable parameter, the corresponding actual-parameter in a procedure-statement or

function-call (see Sections 5.2 and 6.1.2) must be a variable-reference. The formal variable parameter denotes this actual variable during the entire activation of the procedure or function.

Within the procedure or function, any reference to the formal variable parameter is a reference to the actual-parameter itself. The type of the actual-parameter must be *identical* to that of the formal variable parameter. However, if the parameter-type is **string**, then any string-type is considered identical to it; the size attribute of the formal parameter is always the size attribute of the actual parameter.

Note: If the reference to an actual variable parameter involves indexing an array or finding the object of a pointer, these actions are executed before the activation of the procedure or function.

Components of variables of any packed structured-type cannot be used as actual variable parameters.

7.3.3 Procedural Parameters

When the formal-parameter is a procedure-heading, the corresponding actual-parameter in a procedure-statement or function-call (see Sections 5.2 and 6.1.2) must be a procedure-identifier. The identifier in the formal procedure-heading represents the actual procedure during execution of the procedure or function receiving the procedural parameter.

Example of procedural parameters:

```
program PassProc;
  var
    i: integer;

  procedure a( procedure x );
  begin
    write( 'About to call x ' );
    x
  end;

  procedure b;
  begin
    write( 'In procedure b' )
  end;

  function c( procedure x ) : integer;
  begin
    x;
    c:=2
  end;

begin {PassProc}
  a(b);
  i:= c(b)
end.
```

If the formal procedure has a formal-parameter-list, then the actual procedure's declaration must also have a formal-parameter-list and both must be compatible (see Section 7.3.5). However, only the identifier of the actual procedure is written as an actual parameter; no formal or actual parameter-list is given.

Example of procedural parameters with their own formal-parameter-lists:

```
program test;

  procedure xAsPar( y : integer );
  begin
    writeln( 'y=', y )
  end;

  procedure CallProc( procedure xAgain(z:integer) );
  begin
    xAgain(1)
  end;

begin {test}
  CallProc(xAsPar)
end.
```

If the procedural parameter, upon activation, accesses any non-local entity (by variable-reference, procedure-statement, function-call, or label), the entity accessed will be the one that was accessible to the procedure when the procedure was passed as an actual parameter.

To see what this means, consider the following program taken from an example in the ANSI Pascal Standard (which is in turn taken from an early version of the Pascal Validation Suite):

```
program t6p6p3p4( output );
  var
    GlobalOne, GlobalTwo : integer;

  procedure dummy;
  begin
    writeln( 'fail4' )
  end;

  procedure p(procedure f(procedure ff; procedure gg);
             procedure g);
  var
    LocalToP : integer;
```

```

procedure r;
begin
  if GlobalOne = 1 then
    begin
      if (GlobalTwo <> 2) or (LocalToP <> 1) then
        writeln( 'fail1' )
      end
    else if GlobalOne = 2 then
      begin
        if (GlobalTwo <> 2) or (LocalToP <> 2) then
          writeln( 'fail2' )
        else
          writeln( 'pass' )
        end
      else
        writeln( 'fail3' );
        GlobalOne := GlobalOne + 1
      end;
begin {p}
  GlobalTwo := GlobalTwo + 1;
  LocalToP := GlobalTwo;
  if GlobalTwo = 1 then
    p(f,r)
  else
    f(g,r)
  end;

procedure q( procedure f; procedure g );
begin
  f;
  g
end;

begin {program}
  GlobalOne := 1;
  GlobalTwo := 0;
  p( q, dummy )
end.

```

An "execution-trace" of this program yields the following:

1. At the call to *p* in the main program, *GlobalOne*=1 and *GlobalTwo*=0.
2. Within *p*, the formal parameter *f* corresponds to the actual procedure *q* and the formal *g* corresponds to the actual *dummy*. The values of *GlobalTwo* and *LocalToP* both become 1. Because *GlobalTwo*=1, *p* calls itself recursively.
3. Within this second activation of *p*, the formal *f* corresponds to the formal *f* of the first activation, which corresponds to the actual *q*. The formal *g* corresponds to the actual *r*. The values of *GlobalTwo* and *LocalToP* now become 2. Because

GlobalTwo<>1, this second activation of *p* now calls its formal parameter *f*, which is the actual procedure *q*.

4. Within *q*, its formal parameter *f* corresponds to the actual procedure *r* and the formal *g* also corresponds to the actual *r*. Procedure *q* now calls its formals *f* and *g*, i.e. *r* and *r*, and the program terminates after all the various activations unwind.

It is what happens during the two calls to procedure *r* within procedure *q* that is critical. If this program runs correctly, it will print 'pass'. For this to happen, the first call to *r* will occur while *GlobalOne*=1 and will expect *LocalToP* to be 1; the second call to *r* will occur while *GlobalOne*=2 and will expect *LocalToP* to be 2. Since there are no assignments to *LocalToP* within *r* or *q*, how can this be?

LocalToP is not simply local to the procedure *p*, it is local to each activation of *p*. Since there are two activations of *p*, and since *r* is passed as a parameter in each activation of *p*, each *r* accesses the variable *LocalToP* that is local to the activation in which it is passed. Since, in the first activation of *p*, the value of *LocalToP* is 1, that is the value the first execution of *r* sees when it accesses *LocalToP*. Since, in the second activation of *p*, the value of *LocalToP* is 2, that is the value the second execution of *r* sees.

Predefined, inline, and Toolbox procedures can not be passed as procedural parameters.

7.3.4 Functional Parameters

When the formal parameter is a function-heading, the actual-parameter must be a function-identifier. The identifier in the formal function-heading represents the actual function during the execution of the procedure or function receiving the functional parameter.

Functional parameters are exactly like procedural parameters, with the additional rule that corresponding formal and actual functions must have *identical* result-types.

7.3.5 Parameter List Compatibility

Parameter list compatibility is required of the parameter lists of corresponding formal and actual procedural or functional parameters.

Two formal-parameter-lists are compatible if they contain the same number of parameters and if the parameters in corresponding positions match. Two parameters match if one of the following is true:

- They are both value parameters of *identical* type.
- They are both variable parameters of *identical* type.
- They are both procedural parameters with compatible parameter lists.
- They are both functional parameters with compatible parameter lists and result-types.

Section 8

Programs and Units

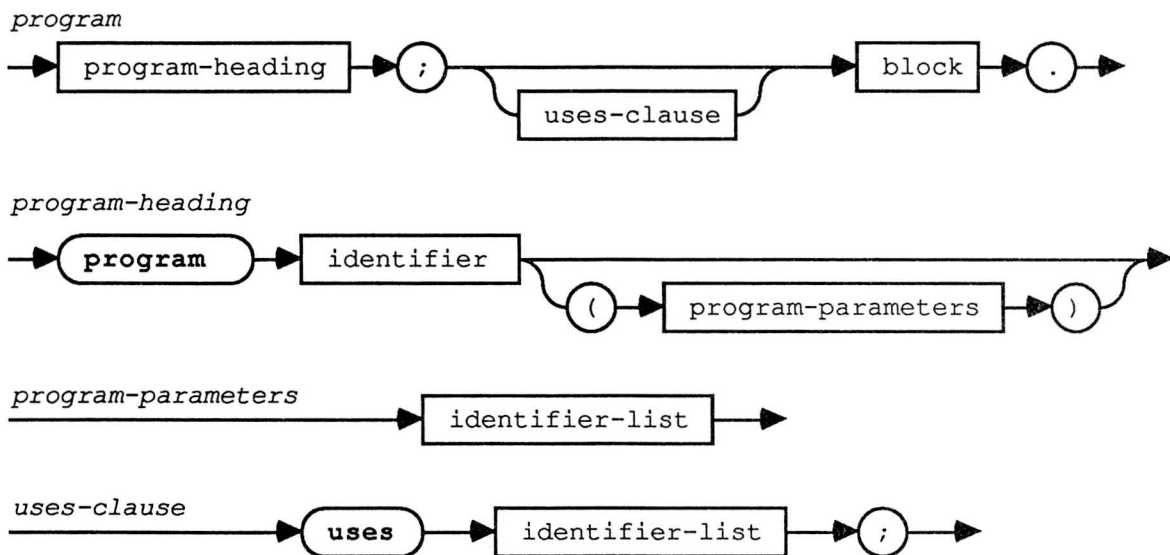
8.1	Program Syntax	89
8.2	Program-Parameters	89
8.3	Unit Syntax	89
8.4	Uses-Clauses	91
8.5	Unit Dependencies	91

Section 8

Programs and Units

8.1 Program Syntax

A Lightspeed Pascal program has the form of a procedure declaration except for its heading.



The occurrence of an identifier immediately after the word **program** declares it as the program's identifier.

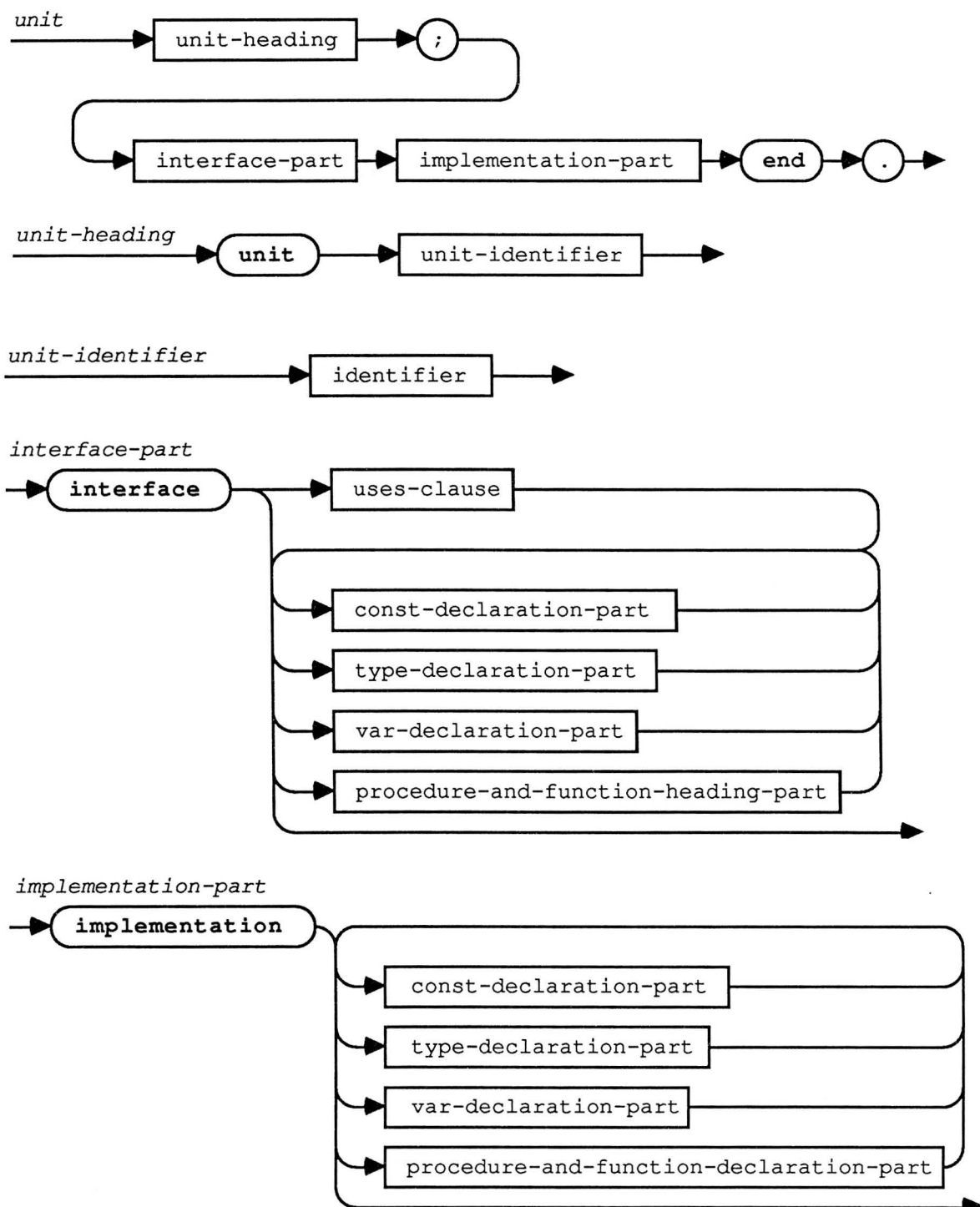
The uses-clause identifies all units required by the program.

8.2 Program-Parameters

Only the predefined identifiers *input* and *output* are allowed as program-parameters.

8.3 Unit Syntax

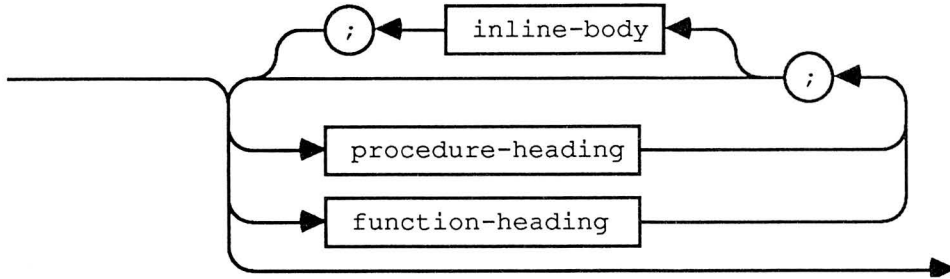
Units provide the means to organize a Pascal program into logically related parts for modular construction of programs and libraries.



The occurrence of an identifier immediately after the word **unit** declares it as the unit's identifier.

The uses-clause identifies all units required by the unit.

procedure-and-function-heading-part



The procedures and functions declared in the interface-part must be redeclared in the implementation-part. The parameters and function types of these redeclarations are omitted, since they were declared in the interface-part. The procedure and function blocks for these routines are included in the implementation-part since they were omitted in the interface-part.

The scope of the declaration for an interface-part is also the implementation-part which is associated with that interface part.

8.4 Uses-Clause

The uses-clause is used to control which units are available to the host (other units or the main program). Each identifier in the identifier-list of the uses-clause is the name of a unit to be made available to the host. All declared entities in the used unit appear as though they are declared in the interface part or the main program block which contains the uses-clause.

8.5 Unit Dependencies

In order to satisfy the requirements of Section 2.2.3, a unit must precede any interface-part or program that it supplies (see Section 2.2.6). It is therefore not possible to construct a valid program in which two units supply each other.

The uses clause in the host must name all units used (directly or indirectly) by the host. Consider the following example:

<pre> program Host; uses UnitA; begin ... end. </pre>	<pre> unit UnitA; interface uses UnitB implementation const a = b; end. </pre>	<pre> unit UnitB; interface const b = 3; implementation ... end. </pre>
---	--	--

The program *Host* uses *UnitA*. *UnitA* uses *UnitB*. There is an identifier *b* defined as a constant in the interface of *UnitB*, but the only reference to *b* is in the implementation part of *UnitA*. In this case, it is not necessary to name *UnitB* in the uses-clause of *Host*.

Now consider the following example:

```
program Host;  
  uses  
    UnitB, UnitA;  
begin  
  ...  
end.  
  
unit UnitA;  
interface  
  uses  
    UnitB;  
  const  
    a = b;  
implementation  
  ...  
end.  
  
unit UnitB;  
interface  
  const  
    b = 3;  
implementation  
  ...  
end.
```

This example is like the previous one, except that this time the reference to the identifier *b* is in the interface part of *UnitA*. In this case, there is an indirect reference to *UnitB* and it is necessary to name *UnitB* in the uses-clause of *Host*. Note that *UnitB* must be named before *UnitA*.

Section 9

Input/Output

9.1	Introduction to I/O	96
9.2	Standard Procedures and Functions for All Files	97
9.2.1	The Reset Procedure	97
9.2.2	The Rewrite Procedure	97
9.2.3	The Open Procedure	98
9.2.4	The Close Procedure	98
9.2.5	The Eof Function	99
9.2.6	The Get Procedure	99
9.2.7	The Put Procedure	100
9.2.8	The Seek Procedure	100
9.2.9	The Filepos Function	100
9.3	Standard Procedures for Non-Textfiles	101
9.3.1	The Read Procedure for Non-Textfiles	101
9.3.2	The Write Procedure for Non-Textfiles	102
9.4	Standard Procedures and Functions for Textfiles	102
9.4.1	The Read Procedure for Textfiles	104
9.4.1.1	Read with a Char-Type Variable	104
9.4.1.2	Read with an Integer-Type Variable	104
9.4.1.3	Read with a Real-Type Variable	105
9.4.1.4	Read with a String-Type Variable	106
9.4.1.5	Read with an Enumerated-Type Variable	106
9.4.2	The Readln Procedure	107
9.4.3	The Write Procedure	108
9.4.3.1	Write-Parameters for Textfiles	108
9.4.3.2	Write with a Char-Type Value	109
9.4.3.3	Write with a String-Type Value	109
9.4.3.4	Write with an Integer-Type Value	109
9.4.3.5	Write with a Real-Type Value	109
9.4.3.6	Write with a Packed-String-Type Value	111
9.4.3.7	Write with an Enumerated-Type Value	111

9.4.4	The Writeln Procedure	111
9.4.5	The Eoln Function	112
9.4.6	The Page Procedure	112
9.4.7	Lazy I/O	113
9.5	Devices on the Macintosh	115

Section 9

Input/Output

This section describes the standard ("built-in") I/O procedures and functions of Lightspeed Pascal.

Standard procedures and functions are predeclared. Since all predeclared entities act as if they were declared in a "block" surrounding the program, no conflict arises from a declaration that redeclares the same identifier within the program.

Note: Standard procedures and functions cannot be used as actual procedural and functional parameters.

Also, the predeclared file variables *input* and *output* do not act as though they are declared in a block outside the program. See Section 9.4.

This section and Section 10 use a modified BNF notation, instead of syntax diagrams, to indicate the syntax of actual-parameter-lists for standard procedures and functions.

Example:

Parameter List: ***write***(*f*, *e*₁ [, *e*₂, ..., *e*_{*n*}])

This represents the syntax of the actual-parameter-list of the standard procedure *write*, as follows:

- *f*, *e*₁, *e*₂, and *e*_{*n*} stand for actual-parameters. Notes on the types and interpretations of the parameters accompany the syntax description.
- The notation *e*₁, *e*₂, ..., *e*_{*n*} means that any number of actual-parameters can appear here, separated by commas.
- Square brackets [] indicate parts of the syntax that can be omitted. They do not indicate sets.

Thus the syntax shown here means that the *f* parameter is required. Any number of *e* parameters may appear, with separating commas, and there must be at least one *e* parameter.

9.1 Introduction to I/O

A Pascal file variable is any variable whose type is a file-type. There are two classes of files: *textfiles* and *non-textfiles*. Any file variable declared to be a type *identical* (see Section 3.5.1) to the standard type *text* is a textfile and all others are non-textfiles. The standard type *text* is roughly equivalent to the type ***packed file of char*** in that a file of type *text* may be treated as though it were a ***packed file of char***. However, the semantics of the two differ somewhat and, in particular, there are certain standard procedures and functions that may be applied to textfiles but not to files of type ***packed file of char***.

A file variable may (but need not) be associated with an external file. The external file may be a named collection of information stored on a peripheral device or, depending on the device, it may be the peripheral device itself. If a file variable is not associated with an external file or device, it is referred to as an *anonymous file*.

For a file variable to be used it must be *opened*. An existing file may be opened via the *reset* and *open* procedures, and a new file may be created and opened via the *rewrite* and *open* procedures. Files opened with *reset* are *read-only* and files opened with *rewrite* are *write-only*. However, *reset* may be applied to a file opened with *rewrite*, which causes the file to become read-only. Likewise, *rewrite* may be applied to a file opened with *reset*, which causes the file to become write-only.

Files opened with *open* are *read/write* files, i.e. they allow both reading and writing. *Reset* and *rewrite* may be applied to files opened with *open*, but they remain read/write files.

The standard file variables *input* and *output*, if present in the program parameter list, are opened automatically when program execution begins and should not be opened again with *reset* or *rewrite*. *Input* is a read-only file associated with your keyboard and *output* is a write-only file associated with the text window.

A file is a linear sequence of *components*, each of which has the component-type of the file. Each component has a *component-number* that is its position in the file relative to the first component in the file. The first component of a file is considered to be component zero.

At any point in time, there is only one component of a file that may be accessed directly through the *file-buffer* denoted by f^{\wedge} . The *current file position* of f is the component number of the component currently accessible through f^{\wedge} . Whenever a file is opened, the current file position is set to component zero, i.e. to the beginning of the file.

Under certain conditions, such as when the current file position is at the end of the file, the value of f^{\wedge} is said to be undefined. It is an error to attempt to use the value of f^{\wedge} when the value is undefined. Assignment to f^{\wedge} is, however, still possible.

Note: It is an error to cause the current file position of a file f to be altered while a reference to the file-buffer f^{\wedge} exists.

Files are normally accessed *sequentially*. That is, when an I/O operation is completed on a file component, the current file position moves to the numerically next file component. Files opened with *open*, however, may also be accessed *randomly* via the standard procedure *seek*, which may be used to specify that the current file position is to be moved to any component number in

the file. The function *filepos*(*f*) may be applied to any file variable *f* and returns the component number of the current file position.

9.2 Standard Procedures and Functions for All Files

9.2.1 The Reset Procedure

Opens an existing file for sequential, read-only access or rewinds an open file.

Parameter List: ***reset*(*f* [, *title*])**

1. *f* is a variable-reference that refers to a variable of file-type. If a *title* is given, the file must not be open. If a *title* is not given, the file must be open.
2. *title* is an optional expression with a string value. The string should be a valid name for a file on a file-structured device, or a name for a non-file-structured device.

Reset(*f*) when *f* is already open causes *f* to be "rewound", i.e. the current file position for *f* is reset to the beginning of the file. If *f* was originally opened with *rewrite*, *f* becomes *read-only*.

Reset(*f*, *title*) finds an existing external file with the name *title*, and associates *f* with this external file. It is an error if there is no existing external file with that name.

The following conditions always hold after *reset*(*f*, [*title*]) is executed:

- *Eof*(*f*) is *true* if the file is empty. Otherwise, *eof*(*f*) is *false*.
- The current file position is the first component of the file (component zero) and the file buffer variable *f*[^] contains the value of that component unless *eof*(*f*) is *true*, in which case the value of *f*[^] is undefined.

9.2.2 The Rewrite Procedure

Creates and opens a new empty file for sequential, write-only access, or rewinds and erases an open file.

Parameter List: ***rewrite*(*f* [, *title*])**

1. *f* is a variable-reference that refers to a variable of file-type. If a *title* is given, the file must not be already open.
2. *title* is an optional expression with a string value. The string should be a valid name for a file on a file-structured device, or a name for a non-file-structured device.

Rewrite(f) (with no *title*) when *f* is not yet open creates an empty anonymous file for writing to.

Rewrite(f) when *f* is already open causes *f* to be "rewound", i.e. the current file position for *f* is reset to the beginning of the file and any prior contents of *f* are deleted. If *f* was originally opened with *reset*, *f* becomes *write-only*.

Rewrite(f, title) creates a new external file with the name *title*, and associates *f* with this external file. If an external file with the name *title* already exists, it is effectively deleted and a new empty file with the same name is created in its place.

The following conditions always hold after *rewrite(f, [title])* is executed:

- *Eof(f)* is *true*.
- The current file position is component zero, i.e. the first component written to the file will become the first component of the file. The value of *f^* is undefined.

9.2.3 The Open Procedure

Opens an existing file or creates and opens a new file for random, read/write access.

Parameter List: ***open(f, title)***

1. *f* is a variable-reference that refers to a variable of file-type. *f* must not be already open.
2. *title* is an expression with a string value. The string should be a valid name for a file on a file-structured device, or a name for a non-file-structured device.

Open(f, title) opens an existing external file with the name *title*, and associates *f* with this external file. If an external file with the name *title* does not already exist, a new empty file is created. The file is opened for both reading and writing.

The following conditions always hold after *open(f, title)* is executed:

- *Eof(f)* is *true* if the file is empty. Otherwise, *eof(f)* is *false*.
- The current file position is component zero and the file buffer variable *f^* contains the value of that component (unless *eof(f)* is *true*).

9.2.4 The Close Procedure

Closes a file.

Parameter List: ***close(f)***

1. *f* is a variable-reference that refers to a variable of file-type. *f* must be open and must not be an anonymous file.

Close(f) closes *f*, i.e. the association between *f* and its external file is broken and the file system marks the external file "closed". All subsequent references to *f* are invalid (except to open it again). In particular, the value of *f*[^] becomes undefined.

If a procedure or function block activation that has a file variable *f* local to it is exited and *f* is not already closed, *f* is closed automatically. If a dynamic variable created with *new* is, or contains, a file variable *f* that is still open when the dynamic variable is destroyed with *dispose*, *f* is closed automatically. If a program terminates with any file still open, the file is automatically closed.

9.2.5 The Eof Function

Detects the end of a file.

Result Type: **boolean**

Parameter List: **eof [(f)]**

1. *f* is a variable-reference that refers to a variable of file-type. If *f* is omitted, the function is applied to the standard file variable *input*. The file must be open.

Eof(f) returns *true* if the current file position is beyond the last component of the file, or if the file contains no components; otherwise, *eof(f)* returns *false*. Specifically, this means the following:

- After a *get*, *eof(f)* returns *true* if the previous file position was the last component of the file.
- After a *put*, *eof(f)* returns *true* if the component written by the *put* is now the last file component.

It is always an error to do a *get(f)* if *eof(f)* is *true*. If *f* is write-only, *eof(f)* will always be *true*.

Note: Whenever *eof(f)* is *true*, the value of the file buffer variable *f*[^] is undefined.

Also, for some devices, *eof* may never be *true*.

9.2.6 The Get Procedure

Advances the current file position and reads the next component of a file.

Parameter List: **get(f)**

1. *f* is a variable-reference that refers to a variable of file-type. The file must be open.

Get (f) advances the current file position to the next file component, and assigns the value of this component to f^{\wedge} . If no next component exists, then *eof (f)* becomes *true*, and the value of f^{\wedge} becomes undefined.

9.2.7 The Put Procedure

Writes the file buffer to the current file position.

Parameter List: **put (f)**

1. *f* is a variable-reference that refers to a variable of file-type. The file must be open and the value of f^{\wedge} must not be undefined.

Put (f) writes the value of f^{\wedge} to *f* at the current file position and advances the current file position to the next file component. If the new file position is beyond the end of the file, *eof (f)* becomes *true*, and the value of f^{\wedge} becomes undefined.

If *eof (f)* is *true*, *put (f)* effectively appends the value of f^{\wedge} to the end of *f* and *eof (f)* remains *true*.

9.2.8 The Seek Procedure

Allows access to an arbitrary file component.

Parameter List: **seek (f, n)**

1. *f* is a variable-reference that refers to a variable of file-type. The file must be open, and it must have been opened with *open*.
2. *n* is an expression with an integer-type value that specifies a file component number in the file. Components in files are numbered from zero.

Seek (f, n) causes the file component numbered *n* to become the current file position. The value of f^{\wedge} becomes the value of that component unless *n* is greater than the number of the last component of the file, in which case *eof (f)* becomes *true* and the value of f^{\wedge} is undefined. Thus, *seek (f, maxlongint)* always sets the current file position to the end of file. *Seek* of a device, such as *Printer:* or *Modem:* is not allowed.

9.2.9 The Filepos Function

Returns the component number of the current file position.

Result Type: **longint**

Parameter List: **filepos (f)**

1. *f* is a variable-reference that refers to a variable of file-type. The file must be open.

Filepos (*f*) returns a *longint* value that is the file component number of the current file position.

9.3 Standard Procedures for Non-Textfiles

The standard procedures in this section may, in fact, be applied to textfiles. However, their interpretation when applied to textfiles is somewhat different and is elaborated in Section 9.4.

9.3.1 The Read Procedure for Non-Textfiles

Reads a file component into a variable.

Parameter List: **read**(*f*, *v*₁ [, *v*₂, ..., *v*_{*n*}])

1. *f* is a variable-reference that refers to a variable of file-type. The file must be open.
2. Each *v* is a variable-reference with a type that the component type of *f* must be assignment-compatible with.

If we consider *ff* to be the variable referenced by *f*, then this form of read is considered to be equivalent to:

```
begin
  read( ff, v1 );
  read( ff, v2 );
  .
  .
  .
  read( ff, vn )
end
```

where *read*(*f*, *v*) is in turn equivalent to:

```
begin
  v := ff^;
  get( ff )
end
```

Note: There is normally a restriction against passing components of packed variables as actual variable parameters (Section 7.3.2). This interpretation of *read* means that each *v* is not considered an actual variable parameter and may therefore be a component of a packed variable.

To understand why the distinction has to be made between *f* and *ff* above, consider the following example:

```

var
    a      : array[ 1..10 ] of file of integer;
    i, j : integer;
    ...
    i := 1;
    read( a[i], i, j );

```

If, say, the value of *i* that is read is 2, and *a[i]* is reevaluated for each *v*, then the value read for *i* will be read from *a[1]* and the value of *j* from *a[2]*. In fact, *a[i]* is evaluated only once before anything is read, and thus all values are read from *a[1]*. *ff* is the result of this one-time evaluation. The use of this *ff* notation will be used again in subsequent sections.

9.3.2 The Write Procedure for Non-Textfiles

Writes a file component from a variable.

Parameter List: **write**(*f*, *e*₁ [, *e*₂, ..., *e*_{*n*}])

1. *f* is a variable-reference that refers to a variable of file-type. The file must be open.
2. Each *e* is an expression with a type that must be assignment-compatible with the component type of *f*.

If we consider *ff* to be the variable referenced by *f*, then this form of *write* is considered to be equivalent to:

```

begin
    write( ff, e1 );
    write( ff, e2 );
    .
    .
    .
    write( ff, en )
end

```

where *write*(*f*, *e*) is in turn equivalent to:

```

begin
    ff^ := e;
    put( ff )
end

```

9.4 Standard Procedures and Functions for Textfiles

This section describes input and output using file variables of the standard type *text*. As previously noted, in Pascal the type *text* is distinct from **packed file of char**. A

textfile is still considered to be a sequence of character components (i.e. is it still a ***packed file of char***). However, it is additionally considered to be a sequence of *lines*, where each line is terminated by an *end-of-line* character.

All of the standard procedures and functions in Section 9.2 may still be applied to a textfile as though it were a ***packed file of char***. However, there are additional procedures and functions that may be applied to textfiles but not other files.

Note: When the value of the file component at the current file position of a file f is an end-of-line character, it appears in the file buffer f^{\wedge} as a space character.

In particular, there are special forms of *read* and *write* that allow you to read and write values that are not of type *char* and will translate them to and from their character representation. For example, *read(f, i)* where i is an *integer* variable will read a sequence of digits (a digit being one of the characters '0' through '9'), interpret that sequence as an integer-type value, and store it in i .

As noted previously there are two standard textfile variables, *input* and *output*. The standard file variable *input* is a read-only file associated with your keyboard. If *input* appears in the program parameter list, then the *input* file is opened automatically when program execution begins as though a *reset* were performed for it. The standard file variable *output* is a write-only file associated with the Text Window. If *output* appears in the program parameter list, then the *output* file is opened automatically when program execution begins as though a *rewrite* were performed for it.

All of the standard procedures and functions in this section need not have a file variable explicitly given as a parameter (in addition to *eof*, as described in Section 9.2.5). In these cases, *input* or *output* will be assumed by default, depending on whether the procedure or function is input-oriented or output-oriented.

- Notes:**
- 1) If the standard file variables *input* and/or *output* are used in a program, their names must appear in the program parameter list. This includes those cases where the standard variables are never explicitly used, but are implied by default.
 - 2) When *input* and/or *output* appear in the program parameter list they may not be redeclared in the main program block. They may, however, be redeclared inside procedure and function blocks.
 - 3) Also, when *input* and *output* are selected by default, the standard file variables are always selected, even though the names *input* and *output* may have been redeclared. User-declared variables with the same names are never selected by default.

9.4.1 The Read Procedure for Textfiles

Reads one or more values from a textfile into one or more program variables.

Parameter List: **read**([*f*,] *v*₁ [, *v*₂, ..., *v*_{*n*}])

1. *f* (if given) is a variable-reference that refers to a variable of type *text*. The file must be open. If *f* is omitted, it is assumed to be the standard *text* file *input*.
2. Each *v* is a variable-reference that refers to a variable of one of the following types:
 - *Char* or a subrange thereof.
 - An integer-type: *integer* (or a subrange thereof) or *longint*.
 - A real-type: *real*, *double*, *extended*, or *computational*.
 - An enumerated-type (including *boolean*) or a subrange thereof.
 - A string-type.

Read(*f*, *v*₁, ..., *v*_{*n*}) is equivalent to:

```
begin
  read( ff, v1 );
  read( ff, v2 );
  ...
  read( ff, vn )
end
```

9.4.1.1 Read with a Char-Type Variable

This is considered equivalent to:

```
begin
  v := ff^;
  get(ff)
end
```

Remember that if the current file position is over an end-of-line character, *ff*[^] contains a space character.

9.4.1.2 Read with an Integer-Type Variable

If *f* is of type *text* and *v* is of an integer-type, then *read*(*f*, *v*) implies the reading from *f* of a sequence of characters that form a signed whole number according to the syntax of Section 1.4

(except that hexadecimal notation is not allowed). If the value read is assignment-compatible with the type of v , then the value is assigned to the variable v ; otherwise, it is an error.

In reading the sequence of characters, blanks and end-of-line characters preceding the first digit or the sign are skipped. Reading ceases as soon as a character is reached that, together with the characters already read, does not form part of a signed whole number, or as soon as $eof(f)$ becomes true.

It is an error if a signed whole number is not found after skipping any preceding blanks and end-of-line characters.

The following things are true immediately after $read(f, v)$ when v is of an integer-type:

- The current file position will be over the character following the last character in the numeric string, unless the last character in the string was the last character in the file.
- $Eof(f)$ will return *true* if the last character in the numeric string was the last character in the file.
- $Eoln(f)$ will return *true* if the last character in the numeric string was the last character on the line.

9.4.1.3 Read with a Real-Type Variable

If f is of type *text* and v is of a real-type, then $read(f, v)$ implies the reading from f of a sequence of characters that represents a signed-number according to the syntax of Section 1.4 (again, except for hexadecimal notation). If the value read is assignment-compatible with the type of v , then the value is assigned to the variable v ; otherwise, it is an error.

In reading the sequence of characters, blanks and end-of-line characters preceding the first digit or the sign are skipped. Reading ceases as soon as a character is reached that, together with the characters already read, does not form a valid signed-number.

It is an error if a valid signed-number is not found after skipping any preceding blanks and end-of-line characters.

Immediately after $read(f, v)$, where v is a real-type variable, the following conditions are true:

- The current file position will be over the character following the last character in the numeric string, unless the last character in the string was the last character in the file.
- $Eof(f)$ will return *true* if the last character in the numeric string was the last character in the file.
- $Eoln(f)$ will return *true* if the last character in the numeric string was the last character on the line.

9.4.1.4 Read with a String-Type Variable

If f is of type *text* and v is of a string-type, then $read(f, v)$ implies the reading from f of a sequence of characters up to *but not including* the next end-of-line character, or until the end of the file. The resulting character-string is assigned to the variable v . It is an error if the number of characters read exceeds the size attribute of v .

Note: *Read* with a string variable does not skip to the next line after reading, and the end-of-line character is left waiting in the file buffer. For this reason, you cannot use successive *read* calls to read a sequence of strings, as they will never get past the first line -- after the first *read*, each subsequent *read* will see the end-of-line and will read a zero-length string. Instead, use *readln* to read string values (see Section 9.4.2). *Readln* skips to the beginning of the next line after reading.

The following things are true immediately after $read(f, v)$ when v is of a string-type:

- The current file position will be over the character following the last character in the string, unless the last character in the string was the last character in the file.
- $Eof(f)$ will return *true* if the last character in the string was the last character in the file.
- $Eoln(f)$ will return *true* unless $eof(f)$ is *true*.

9.4.1.5 Read with an Enumerated-Type Variable

If f is of type *text* and v is of an enumerated type, then $read(f, v)$ implies the reading from f of a sequence of characters that form an identifier according to the syntax of Section 1.2. If the identifier read is identical (ignoring the case of letters) to an enumerated constant of the enumerated type of v , the value of the enumerated constant is assigned to v ; otherwise, it is an error.

In reading the sequence of characters, blanks and end-of-line characters preceding the first letter of the identifier are skipped. Reading ceases as soon as a character is reached that, together with the characters already read, does not form part of an identifier, or as soon as $eof(f)$ becomes true.

It is an error if an identifier is not found after skipping any preceding blanks and end-of-line characters.

If f is of type *text*, the following things are true immediately after $read(f, v)$ when v is an enumerated-type variable:

- The current file position will be over the character following the last character in the identifier, unless the last character in the string was the last character in the file.
- $Eof(f)$ will return *true* if the last character in the identifier was the last character in the file.

- *Eoln(f)* will return *true* if the last character in the identifier was the last character on the line.

9.4.2 The Readln Procedure

The *readln* procedure is an extension of *read* for textfiles. Essentially it does the same thing as *read*, and then skips to the beginning of the next line in the input file.

Parameter List: same as for *read*, except as follows:

- A *readln* call with no input variables is allowed, e.g.

```
readln(sourcefile)
```

- The parameter-list can be omitted altogether.

If the first parameter does not specify a file variable, or if the parameter-list is omitted, the procedure reads from the standard file *input*.

Readln(f), with no input-variables, causes the current file position to advance to the beginning of the next line (if there is one, else to the end of the file), i.e.:

```
begin
  while not eof(ff) and not eoln(ff) do
    get(ff);
  if not eof(ff) then
    get(ff)
end
```

Readln(f, v₁, ..., v_n) is equivalent to:

```
begin
  read(ff, v1, ..., vn);
  readln(ff)
end
```

The following are true immediately after *readln(f, v)*, regardless of the type of *v*:

- *Eof(f)* will return *true* if the line read was the last line in the file.
- *Eoln(f)* will return *false* unless the line following the line read is empty.

9.4.3 The Write Procedure for Textfiles

Writes one or more values to a *text* file.

Parameter List: **write**([*f*,] *p*₁ [, *p*₂, ..., *p*_{*n*}])

1. *f* (if given) is a variable-reference that refers to a variable of type *text*. The file must be open. If *f* is omitted, the procedure writes to the standard file *output*.
2. *p*₁, ..., *p*_{*n*} are *write-parameters*. Each write-parameter includes an *output expression*, whose value is to be written to the file. As explained below, a write-parameter may also contain the specifications of a field-width and a number of decimal places. Each output expression must have a result of char-type, an integer-type, a real-type, a string-type, a packed-string-type, or an enumerated-type. At least one write-parameter must be present.

Write(*f*, *p*₁, ..., *p*_{*n*}) is equivalent to:

```
begin
  write( ff, p1 );
  ...
  write( ff, pn )
end
```

9.4.3.1 Write-Parameters

Each write-parameter has the form

OutExpr [: *MinWidth* [: *DecPlaces*]]

where *OutExpr* is an output expression. *MinWidth* and *DecPlaces* are expressions with integer-type values.

MinWidth specifies the *minimum* field width. *MinWidth* must be greater than zero. Exactly *MinWidth* characters are written (using leading spaces if necessary), except when *OutExpr* has a value that must be represented in more than *MinWidth* characters; in this case, enough characters are written to represent the value of *OutExpr*. Likewise, if *MinWidth* is omitted, then enough characters as necessary are written to represent the value of *OutExpr*.

DecPlaces specifies the number of decimal places in a fixed-point representation of a *real* value. It can be specified only if *OutExpr* has a real-type value, and if *MinWidth* is also specified. If specified, it must be greater than zero. If *DecPlaces* is not specified, a floating-point representation is written.

9.4.3.2 Write with a Char-Type Value

If *MinWidth* is omitted, the character value of *OutExpr* is written on the file *f*. Otherwise, *MinWidth*-1 spaces followed by the character value of *OutExpr* is written.

9.4.3.3 Write with a String-Type Value

Assuming the string value of *OutExpr* has a length *L*, if $L < \text{MinWidth}$, the string value is written on the file *f* preceded by *MinWidth*-*L* spaces. If $L > \text{MinWidth}$, the first *MinWidth* characters of the string are written. If $L = \text{MinWidth}$, or if *MinWidth* is omitted, the entire string value is written on the file.

9.4.3.4 Write with an Integer-Type Value

If *OutExpr* has an integer-type value, its decimal (base 10) representation is written on the file *f*. Assume that *OutDigits* is a string-type value that contains the decimal representation of $\text{abs}(\text{OutExpr})$ with no leading zeroes unless the value of *OutExpr*=0, in which case *OutDigits* contains the single character "0". If *MinWidth* is omitted from the write-parameter, then it is assumed to be zero. Thus, the representation of *OutExpr* is written to *f* as if by the algorithm:

```
begin
  if MinWidth >= length(OutDigits)+1 then
    write( ff, ' ' : MinWidth-length(OutDigits)-2 );
  if OutExpr < 0 then
    write( ff, '-' )
  else if MinWidth >= length(OutDigits)+1 then
    write( ff, ' ' );
  write( ff, OutDigits )
end
```

9.4.3.5 Write with a Real-Type Value

If *OutExpr* has a real-type value, its decimal representation is written on the file *f*. This representation depends on the presence or absence of and, if present, the value of *DecPlaces*.

If *DecPlaces* is present, a *fixed-point* representation is written. Assume that *IntDigits* is a string-type value that contains the decimal representation of $\text{trunc}(\text{abs}(\text{OutExpr}))$ with no leading zeroes unless the value of *OutExpr*=0, in which case *IntDigits* contains the single character "0". Assume that *FracDigits* is a string-type value that contains the decimal representation of

$$\text{round}(\text{abs}(\text{OutExpr}) - \text{trunc}(\text{abs}(\text{OutExpr})) * 10^{\text{DecPlaces}})$$

with enough leading zeroes to make $\text{length}(\text{FracDigits}) = \text{DecPlaces}$. Thus, the fixed-point representation is written to *f* as if by the algorithm:

```

begin
  if MinWidth>=length(IntDigits)+length(FracDigits)+2 then
    write( ff, ' ' : MinWidth-TotalDigits-3 );
  if OutExpr<0 then
    write( ff, '-' )
  else
    if MinWidth>=length(IntDigits)+length(FracDigits)+2 then
      write( ff, ' ' );
    write( ff, IntDigits, '.', FracDigits )
  end
end

```

If *DecPlaces* is not specified, a *floating-point* representation is written. If *MinWidth* is omitted from the write-parameter, then it is assumed to be 10. Assume that *abs (OutExpr)* has a representation in the floating-point notation of the form:

$$m.n \times 10^e$$

where $0 < m \leq 9$ unless *OutExpr*=0, in which case $m=n=e=0$. Assume that *IntDigit* is a string-type value that contains the decimal representation of *m* (a single digit). Assume that *FracDigits* is a string-type value that contains the first *MinWidth*-9 digits of the decimal representation of *n* rounded, and with leading zeroes retained and trailing zeroes added if necessary. Assume that *ExpDigits* is a string-type value that contains the decimal representation of *abs (e)* with enough leading zeroes to make *length (ExpDigits)*=4. Also assume that *NegExp* has the value *true* if $e < 0$, and otherwise the value *false*. Thus, the floating-point representation is written to *f* as if by the algorithm:

```

begin
  if OutExpr<0 then
    write( ff, '-' )
  else
    write( ff, ' ' );
  write( ff, IntDigit, '.', FracDigits, 'E' );
  if NegExp then
    write( ff, '-' )
  else
    write( ff, '+' );
  write( ff, ExpDigits )
end

```

9.4.3.6 Write with a Packed-String-Type Value

If *OutExpr* is of a packed-string-type, the effect is the same as writing a string whose length is the number of components in the type.

9.4.3.7 Write with an Enumerated-Type Value

If the value of *OutExpr* is of an enumerated type, the string representation of the enumerated constant identifier corresponding to the value is written on the file *f*. If the length of this string representation is *L* and $L < \text{MinWidth}$, then $\text{MinWidth} - L$ spaces are written out before the string. In any case the entire string is always written, even if $L > \text{MinWidth}$.

9.4.4 The Writeln Procedure

The *writeln* procedure is an extension of *write* for textfiles. Essentially it does the same thing as *write*, and then writes an end-of-line character to the output file (ending the line).

**Parameter List:* same as for *write*, except as follows:

- A *writeln* call with no write-parameters is allowed. *Example:*

```
writeln(outputfile)
```

- The parameter-list can be omitted altogether.

If the first parameter does not specify a file variable, or if the parameter-list is omitted, the procedure writes to the standard file *output*.

Writeln(f) writes an end-of-line character to the file *f*.

Writeln(f, p₁, ..., p_n) is equivalent to:


```

begin
    write( ff, p1, ..., pn );
    writeln(ff)
end

```

The following are true immediately after `writeln(f, v)`, regardless of the type of `v`:

- `Eof(f)` will return *true* if the last character written became the last character in the file. If `f` is write-only, then `eof(f)` will necessarily be *true*.
- `Eoln(f)` will return *false* unless the character following the last character written is an end-of-line character.

9.4.5 The Eoln Function

Result Type: **boolean**

Parameter List: **eoln [(f)]**

1. `f` is a variable-reference that refers to a variable of type *text*. The file must be open. If `f` is omitted, the function is applied to the standard file *input*.

`Eoln(f)` returns *true* if the character at the current file position is an end-of-line character. It is an error to call `eoln(f)` if `f` is a non-textfile, if `f` is write-only, or if `eof(f)` is *true*.

Note: Every line in a file is expected to be terminated by an end-of-line character. This may not actually be the case, i.e. the last character in a file may not be an end-of-line character as it should. If a file `f` is read-only (i.e. opened with *reset*) then, upon reaching the end of the file, if the last character was not an end-of-line character, `f^` becomes a space character, `eoln(f)` becomes *true*, and `eof(f)` remains *false*. The next attempt to read a character will then cause `eof(f)` to become *true*. This will only happen if the file is read-only and not if it is read/write.

9.4.6 The Page Procedure

Parameter List: **page [(f)]**

1. `f` is a variable-reference that refers to a variable of type *text*. The file must be open.

If `f` is omitted, the standard textfile *output* is assumed. `Page(f)` causes a skip to the top of a new page when `f` is printed or displayed. If `f` is write-only, and if the last character in `f` is not an end-of-line character, then one is inserted before the *page* is done.

9.4.7 Lazy I/O

Consider the following (fairly trivial) program:

```
program count( input, output );  
  var  
    s  : string;  
    ch : char;  
begin  
  write( 'Type a line of characters -- ' );  
  readln(s);  
  writeln( 'You typed ', length(s),  
           ' characters on that line' );  
end.
```

If all of the preceding sections are taken literally, then there are two problems with this program:

1. Because *reset* (which is done implicitly for *input* when the program starts) causes the first character of *input* to appear in *input*[^], the program will hang waiting for input from the keyboard before the *write* statement is executed. This means the prompt the program is supposed to give you for input will not appear until after you type a character.
2. Having typed a line of characters followed by the *return key* (the "end-of-line key" so to speak), *readln* causes the first character of the line following the one just read to appear in *input*[^]. This means that the *writeln* following the *readln* will not be executed until you type another character following the return.

This behavior is a *malaise* of Pascal's that has been the bane of Pascal programmers ever since the language was created. It exists (in part) because Pascal was originally designed to run on a *batch* computer system (at a time when *interactive*, terminal-driven systems were rare).

On such a system, *input* was expected to be associated with a previously prepared input file (typically a deck of punched cards) and *output* was expected to be associated with a file where the results of your program would appear for your inspection after its execution was complete (typically a line printer listing).

Although, when the ANSI Pascal Standard was being drafted, many subtle changes were made to the Pascal language, yet somehow this problem was never completely resolved. So many programs had already been written that depended on this behavior when doing I/O. To change the language in any significant way would have made these programs invalid. Instead, the Standard is worded in such a way that it is *possible* to get around this problem.

For instance, when the standard specifies that, after *reset (f)*, *f*[^] contains the first component of the file, it does so in a way that allows *f*[^] to remain undefined until its value is needed. Thus, in the above program, it is not strictly necessary to read the first character from the keyboard as soon as execution begins; it suffices to do so when the *readln* needs that character. Likewise, after *readln* processes the end-of-line character (the return key), it is not necessary to then read another character from the keyboard. It is in fact never necessary because the program does not reference *input* again.

Interpreting the Standard's semantics in this way is a technique that is popularly known as *Lazy I/O*, and is the only known technique that allows interactive I/O in Pascal while at the same time preserving the Standard's I/O semantics. Other techniques exist, but they cause I/O operations to behave differently depending on whether you have specified that the I/O is to be done interactively or not. It is thus difficult, with these techniques, to write a program that works the same when it is reading from a file as when it is reading from a keyboard.

Specifically, using the Lazy I/O technique involves separating the operations of advancing the current file position and doing input from the file by *deferring* the actual input of data from a file until absolutely necessary. Conditions that make input from a file f necessary include:

1. A reference to f^{\wedge} other than to assign it a value or to pass it as an actual variable parameter.
2. A call to $eof(f)$. In this case, input may be necessary because it is not (necessarily) possible to know whether the end of the file has been reached without trying to read beyond it.
3. A call to $eoln(f)$. In this case, input may be necessary to get another character and see if it is an end-of-line character.
4. A call to $get(f)$. In this case, input will only be necessary if prior to the call to get there was deferred input that was never actually done. If so, the original deferred input will be performed, but the input implied by the get will in turn be deferred.
5. A call to $read$ or $readln$ from f . Here, the input necessary to do the $read$ or $readln$ will be performed, but the input of the component following the last component read will be deferred.

Although Lazy I/O makes it possible to write interactive programs without much difficulty, you should nevertheless be aware of the above to avoid peculiar situations that may cause your program to hang waiting for input unexpectedly. As an example:

```
program process_lines( input, output );
var
  s : string;
begin
    repeat
      write('>');
      readln(s);
      ...
    until eoln
end.
```

Such a program might be intended to read and process lines, prompting each line with the '>' character, until an empty line is entered. An empty line is an end-of-line character immediately following the end of the previous line. Thus, the above program reads a line and tests to see if an end-of-line immediately follows. The problem is that the program, after prompting for and reading the first line, will as a result of the $eoln$ stop and wait for another character to be typed

before issuing another prompt. A better way to write this program, one that avoids this problem, might be:

```
program process_lines( input, output );  
  var  
    s      : string;  
    empty : boolean;  
begin  
  empty := false;  
  repeat  
    write('>');  
    if eoln then  
      empty := true  
    else  
      begin  
        readln(s);  
        ...  
      end  
    until empty  
end.
```

or even better:

```
program process_lines( input, output );  
  var  
    s : string;  
begin  
  repeat  
    write('>');  
    readln(s);  
    ...  
  until length(s) = 0  
end.
```

9.5 Devices on the Macintosh

On the Macintosh, there are basically three devices to be concerned about: disk drives, a printer, and a modem.

The disk drives are never addressed directly. Rather, you address the disks themselves by name. Each disk is referred to as a *volume* and the disk's name is its *volume name*. For disk files, the *title* parameter for *reset*, *rewrite*, and *open* (see Section 9.2.1 through 9.2.3) consists of a file name optionally preceded by a volume name and a colon, e.g.:

MyVolume:MyFile

More information on volumes and files can be found in the Macintosh system documentation on the Macintosh *File Manager*.

For the printer and modem, the title parameter for *reset*, *rewrite*, and *open* consists of the device name followed by a colon, e.g.:

Printer:

[A file name may follow the colon, but it will be ignored.]

The device name for the printer is simply *printer*. Likewise the device name for the modem is *modem*.

Warning: Since the printer is a write-only device, you can only use '*printer:*' as the title parameter for *rewrite*; it is an error to give '*printer:*' as the title parameter for *reset* or *open*.

A file variable opened with '*modem:*' as the title parameter reads from and writes to the modem at 300 baud. No 'EOF' character is defined for the '*modem:*' device; it is therefore meaningless to use the *EOF* predefine to test for end-of-file on a '*modem:*' device.

It is an error to open a file variable with '*printer:*' or '*modem:*' as the title parameter if it is not of type *text*.

For the printer and/or the modem to work properly, they must be connected to their proper sockets in the back of the Macintosh (see the Macintosh user's manual for details).

The standard Text Window is treated as a write-only device with the name *TextWindow:*. It can be used only with the predefined file *Output*, implying that *Output* must first be closed.

The keyboard is treated as a read-only device with the name *Keyboard:*. It can only be used with the predefined file *Input*, implying that *Input* must first be closed.

Section 10

Standard Procedures and Functions

10.1	Dynamic Allocation Procedures	119
10.1.1	The New Procedure	119
10.1.2	The Dispose Procedure	120
10.2	Transfer Procedures and Functions	121
10.2.1	The Pack Procedure	121
10.2.2	The Unpack Procedure	121
10.2.3	The Trunc Function	122
10.2.4	The Round Function	122
10.2.5	The Ord4 Function	122
10.2.6	The Pointer Function	123
10.3	Arithmetic Functions	123
10.3.1	The Odd Function	123
10.3.2	The Abs Function	124
10.3.3	The Sqr Function	124
10.3.4	The Sqrt Function	124
10.3.5	The Sin Function	125
10.3.6	The Cos Function	125
10.3.7	The Exp Function	125
10.3.8	The Ln Function	125
10.3.9	The Arctan Function	126
10.4	Ordinal Functions	126
10.4.1	The Ord Function	126
10.4.2	The Chr Function	126
10.4.3	The Succ Function	127
10.4.4	The Pred Function	127
10.5	String Procedures and Functions	127
10.5.1	The Length Function	127
10.5.2	The Pos Function	128
10.5.3	The Concat Function	128
10.5.4	The Copy Function	128

10.5.5	The Delete Procedure	129
10.5.6	The Omit Function	129
10.5.7	The Insert Procedure	129
10.5.8	The Include Function	130
10.6	Lightspeed Pascal Window Manipulation Procedures	130
10.6.1	The HideAll Procedure	130
10.6.2	The ShowText Procedure	130
10.6.3	The ShowDrawing Procedure	131
10.6.4	The SetTextRect Procedure	131
10.6.5	The SetDrawingRect Procedure	131
10.6.6	The GetTextRect Procedure	131
10.6.7	The GetDrawingRect Procedure	131
10.6.8	The SaveDrawing Procedure	132
10.7	Miscellaneous Procedures and Functions	132
10.7.1	The Sizeof Function	132
10.7.2	Logical Operations	132
10.7.3	Operations on Longint Values	133
10.7.4	The WriteDraw Procedure	133
10.7.5	The StringOf Function	134
10.7.6	The ReadString Procedure	134
10.7.7	The OldFileName Function	135
10.7.8	The NewFileName Function	135
10.7.9	The Synch Procedure	135
10.7.10	The Note Procedure	136
10.7.11	The InLine Procedures	136
10.7.12	The Generic Procedure	137
10.7.13	The Macsbug Support Procedures	138

Section 10

Standard Procedures and Functions

This section describes all the standard ("built-in") procedures and functions in Lightspeed Pascal, except for the I/O procedures and functions described in Section 9, and the Macintosh Toolbox procedures and functions described in Appendices C and E.

Standard procedures and functions are predeclared. Since all predeclared entities act as if they were declared in a block surrounding the program, no conflict arises from a declaration that redeclares the same identifier within the program.

Note: Standard procedures and functions cannot be used as actual procedural and functional parameters.

This section uses a modified BNF notation, instead of syntax diagrams, to indicate the syntax of actual-parameter-lists for standard procedures and functions. The notation is explained at the beginning of Section 9.

10.1 Dynamic Allocation Procedures

The procedure *new* is used for the creation of dynamic variables by the program. Dynamic variables are variables that can only be accessed through pointer variables, and they are created by allocating a region of memory from a larger portion of free memory called the *heap*. The address of the allocated region is the pointer value that is used to access the dynamic variable. The *dispose* procedure is used to destroy dynamic variables created with *new*, in the process returning that variable's region of memory to the heap for reuse.

10.1.1 The New Procedure

Creates a new dynamic variable and sets a pointer variable to point to it.

Parameter List: ***new***(*p* [, *c*₁, *c*₂, ..., *c*_{*n*}])

1. *p* is a variable-reference that refers to a variable of any pointer-type. This is a variable parameter.
2. Each *c* (if given) is a constant. If the base-type of *p* is a record-type with variants, then each *c* may be a constant corresponding to a case-constant of a variant of a variant-part (see below).

New(*p*) creates a new variable of the base-type of *p*, and makes *p* point to it. The variable can be referenced as *p*^.

It is an error if the heap does not contain enough free space to create the new variable.

Normally, if the base-type of *p* is a record-type with a variant-part, *new*(*p*) allocates enough memory to the variable to accommodate the largest variant. However, by supplying a constant *c* that is a case-constant of one of the variants of the variant-part, only the amount of memory necessary to accommodate that particular variant is allocated to the variable.

If the record variant itself contains a variant-part, then a case-constant *c* may be given to select a particular variant of that variant-part, and so on. Each *c* in the parameter list must be given in the order of the nesting of the variants, one for each level of nesting.

If the variant selected by *c_n* (the last *c* given in the parameter list) has itself a variant-part, then enough memory is allocated to the variable to accommodate the largest variant of that variant-part.

Warning: When a record variable is dynamically allocated with case-constants as shown above, you must not make assignments to any fields of variants that are not selected by the case-constants. Also, you must not assign an entire record variable to this record or use the entire record in an expression or as an actual parameter.

10.1.2 The Dispose Procedure

Destroys a dynamic variable.

Parameter List: ***dispose***(*p* [, *c*₁, *c*₂, ..., *c*_{*n*}])

1. *p* is a variable-reference that refers to a pointer-variable. It must be a pointer that was previously assigned by the *new* procedure or was assigned a meaningful value by an assignment statement. It is an error to attempt to *dispose* of a pointer variable that is currently being accessed, or whose value is undefined or is ***nil***.
2. Each *c* (if given) is a constant. If the base-type of *p* is a record-type with variants, then each *c* may be a constant corresponding to a case-constant of a variant of a variant-part (see 10.1.1 above).

Dispose(*p*) destroys the variable referenced by *p* and returns its memory region to the heap. The value of *p* then becomes undefined and it is an error to subsequently make reference to *p*^.

If the dynamic variable pointed to by *p* was created by *new* with a list of case-constants, then the same list of case-constants (in the same order) must be given to *dispose*.

10.2 Transfer Procedures and Functions

10.2.1 The Pack Procedure

Transfers the contents of an **array** to a **packed array**.

Parameter List: **pack(a, i, z)**

1. *a* is a variable of an array-type and *z* is a variable of a packed-array-type whose component-type is the same as the component-type of *a*. *i* is an expression whose value is assignment-compatible with the index-type of *a*.

Pack(a, i, z) copies the components of *a* to *z* starting at the i^{th} component of *a* and continues until all components of *z* have been copied to.

If *j* is a variable with the same type as the index-type of *z*, if *k* is a variable with the same type as the index-type of *a*, and if *u* and *v* are the lowest and highest possible values of the index-type of *z*, then *pack(a, i, z)* is equivalent to:

```
begin
  k := i;
  for j := u to v do
    begin
      z[j] := a[k];
      if j <> v then
        k := succ(k)
    end
  end
```

It is an error if the number of components comprising the i^{th} through last components of *a* is less than the number of components of *z*. It is an error if any component of *a* accessed is undefined.

10.2.2 The Unpack Procedure

Transfers the contents of a **packed array** to an **array**.

Parameter List: **unpack(z, a, i)**

1. *z* is a variable of a packed array-type and *a* is a variable of an array-type whose component-type is the same as the component-type of *z*. *i* is an expression whose value is assignment-compatible with the index-type of *a*.

Unpack(z, a, i) copies the components of *z* to *a* starting at the i^{th} component of *a* and continues until all components of *z* have been copied.

If *j* is a variable with the same type as the index-type of *z*, if *k* is a variable with the same type as the index-type of *a*, and if *u* and *v* are the lowest and highest possible values of the index-type of *z*, then *unpack(z, a, i)* is equivalent to:

```

begin
  k := i;
  for j := u to v do
    begin
      a[k] := z[j];
      if j <> v then
        k := succ(k)
      end
    end
  end

```

It is an error if the number of components comprising the i^{th} through last components of a is less than the number of components of z . It is an error if any component of z is undefined.

10.2.3 The Trunc Function

Converts a real-type value to a *longint* value.

Result Type: *longint*

Parameter List: **trunc(x)**

1. x is an expression with a value of a real-type.

Trunc(x) returns a *longint* result that is the value of x rounded to the nearest whole number that is between 0 and x inclusive. It is an error if the result of this rounding is outside the range $-\text{maxlongint}..\text{maxlongint}$.

10.2.4 The Round Function

Converts a real-type value to a *longint* value.

Result Type: *longint*

Parameter List: **round(x)**

1. x is an expression with a value of a real-type.

Round(x) returns a *longint* result that is the value of x rounded to the nearest whole number. If x is exactly halfway between two whole numbers, the result is the whole number with the greatest absolute magnitude. It is an error if the result of this rounding is outside the range $-\text{maxlongint}..\text{maxlongint}$.

10.2.5 The Ord4 Function

Converts an ordinal-type or pointer-type value to a *longint* value.

Result Type: *longint*

Parameter List: **ord4(x)**

1. x is an expression with a value of ordinal-type or pointer-type.

Ord4 (x) returns the ordinal value of x .

If x is of a pointer-type, the result is the address of the dynamic variable pointed to by x .

If x is of an ordinal-type, the result is the ordinality of x (see Section 3.1.1), represented as a *longint*.

10.2.6 The Pointer Function

Converts an integer-type value to a generic pointer-type value.

Result Type: a generic pointer which matches any pointer

Parameter List: ***pointer***(x)

1. x is an expression with a value of integer-type.

Pointer(x) returns a pointer value that points to whatever is at the address x as though it were a dynamic variable created at that address. This pointer is of the same type as ***nil*** in that it is assignment-compatible with any pointer-type.

As a convenience, pointer may be also applied to an expression of any pointer-type, effectively making that expression assignment-compatible with any (other) pointer-type.

10.3 Arithmetic Functions

In general, any *extended* real-type result returned by an arithmetic function is an approximation. There is one exception to this: the result of the *abs* function is exact.

10.3.1 The Odd Function

Tests whether an integer-type value is odd.

Result Type: *boolean*

Parameter List: ***odd***(x)

1. x is an expression with a value of an integer-type.

Odd(x) returns *true* if x is odd, i.e. not divisible by 2 without a remainder. If x is even it returns *false*.

10.3.2 The Abs Function

Returns the absolute value of a numeric value.

Result Type: integer, longint, or extended.

*Parameter List: **abs**(*x*)*

1. *x* is an expression with a value of an integer-type or a real-type.

Abs(*x*) returns the absolute value of *x*; i.e. if *x* is negative, $-x$ is returned; otherwise *x* is returned. If *x* is of a real-type, the result type is *extended*. If *x* is of type *longint*, the result type is *longint*. Otherwise, the result type is *integer*.

10.3.3 The Sqr Function

Returns the square of a numeric value.

Result Type: integer, longint, or extended.

*Parameter List: **sqr**(*x*)*

1. *x* is an expression with a value of an integer-type or a real-type.

Sqr(*x*) returns the square of *x*, i.e. $x*x$.

It is an error if the result is not within the range of values representable by the result-type (see Section 3.1).

10.3.4 The Sqrt Function

Returns the square root of a numeric value.

Result Type: extended

*Parameter List: **sqrt**(*x*)*

1. *x* is an expression with a value of an integer-type or real-type. It is an error if $x < 0$.

Sqrt(*x*) returns the positive square root of *x*, i.e. the positive value *y* such that $y*y=x$. It is an error if the result is a value too small to be represented by the real-type *extended* (see Section 3.1.2).

10.3.5 The Sin Function

Returns the sine of a numeric value.

Result Type: extended

*Parameter List: **sin(x)***

1. x is an expression with a value of an integer-type or real-type. This value is assumed to represent an angle in radians.

Sin(x) returns the trigonometric sine of x .

10.3.6 The Cos Function

Returns the cosine of a numeric value.

Result Type: extended

*Parameter List: **cos(x)***

1. x is an expression with a value of an integer-type or real-type. This value is assumed to represent an angle in radians.

Cos(x) returns the trigonometric cosine of x .

10.3.7 The Exp Function

Returns the exponential of a numeric value.

Result Type: extended

*Parameter List: **exp(x)***

1. x is an expression with a value of an integer-type or real-type.

Exp(x) returns the value of e^x , where e is the base of the natural logarithms. It is an error if the result cannot be represented with the real-type *extended* (see Section 3.1.2).

10.3.8 The Ln Function

Returns the natural logarithm of a numeric value.

Result Type: extended

*Parameter List: **ln(x)***

1. x is an expression with a value of an integer-type or real-type. It is an error if $x \leq 0$.

$\text{Ln}(x)$ returns the natural logarithm (\log_e) of x .

10.3.9 The Arctan Function

Returns the arctangent of a numeric value.

Result Type: *extended*

Parameter List: **arctan(x)**

1. x is an expression with a value of an integer-type or real-type. It is an error if $x < 0$.

$\text{Arctan}(x)$ returns the principal value, in radians, of the arctangent of x .

10.4 Ordinal Functions

10.4.1 The Ord Function

Returns the ordinal number of an ordinal-type or pointer-type value.

Result Type: *integer* or *longint*

Parameter List: **ord(x)**

1. x is an expression with a value of ordinal-type or pointer-type.

If x is of pointer-type, the result is the *longint* address of the dynamic variable pointed to by x .

If x is of an ordinal-type, the result type is the ordinality of x (see Section 3.1.1). If x is of type *longint*, the result type is *longint*. Otherwise, the result type is *integer*.

10.4.2 The Chr Function

Returns the *char* value corresponding to a whole-number value.

Result Type: *char*

Parameter List: **chr(x)**

1. x is an expression with an integer-type value that must be in the range 0..255.

$\text{Chr}(x)$ returns the *char* value whose ordinal number is x (see Appendix J).

For any *char* value ch , the following is always true:

$$\text{chr}(\text{ord}(ch)) = ch$$

10.4.3 The Succ Function

Returns the successor of a value of ordinal-type.

Result Type: same as parameter

Parameter List: **succ(*x*)**

1. *x* is an expression with a value of ordinal-type.

Succ(x) returns the successor of *x*.

It is an error if *x* is the last value in the type of *x*, i.e. it has no successor. Otherwise

$$\text{ord}(\text{succ}(x)) = \text{ord}(x) + 1$$

10.4.4 The Pred Function

Returns the predecessor of a value of ordinal-type.

Result Type: same as parameter

Parameter List: **pred(*x*)**

1. *x* is an expression with a value of ordinal-type.

Pred(x) returns the predecessor of *x*.

It is an error if *x* is the first value in the type of *x*, i.e. it has no predecessor. Otherwise,

$$\text{ord}(\text{pred}(x)) = \text{ord}(x) - 1$$

10.5 String Procedures and Functions

10.5.1 The Length Function

Returns the current length of a value of string-type.

Result Type: integer

Parameter List: **length(*str*)**

1. *str* is an expression with a value of a string-type.

Length(str) returns the current length attribute of *str* (see Section 3.3).

10.5.2 The Pos Function

Searches a string for the first occurrence of a specified substring.

Result Type: integer

*Parameter List: **pos(substr, str)***

1. *substr* is an expression with a value of a string-type.
2. *str* is an expression with a value of a string-type.

Pos(substr, str) searches for *substr* within *str*, and returns an *integer* value that is the index of the first character of *substr* within *str*.

If *substr* is not found, *pos(substr, str)* returns zero.

10.5.3 The Concat Function

Takes a sequence of strings and concatenates them.

Result Type: string-type

*Parameter List: **concat(str₁ [, str₂, ..., str_n])***

- Each parameter is an expression with a value of string-type. Any practical number of parameters may be passed.

Concat(str₁, ..., str_n) concatenates all the parameters in the order in which they are written, and returns the concatenated string. Note that the number of characters in the result cannot exceed 255.

10.5.4 The Copy Function

Returns a substring of specified length, taken from a specified position within a string.

Result Type: string-type

*Parameter List: **copy(source, index, count)***

1. *source* is an expression with a value of a string-type.
2. *index* is an expression with an integer-type value.
3. *count* is an expression with an integer-type value.

Copy(source, index, count) returns a string containing *count* characters from *source*, beginning at *source[index]*. If *count* ≤ 0, then a null string is returned. If *index* < 1 and/or *index+count* > *length(source)*, i.e. if character positions outside the

range $1..length(source)$ are implicitly referenced, it is not an error. However, only the characters that lie within that range are copied.

10.5.5 The Delete Procedure

Deletes a substring of specified length from a specified position within the value of a string variable.

Parameter List: **`delete(dest, index, count)`**

1. *dest* is a variable-reference that refers to a variable of a string-type. This is a variable parameter.
2. *index* is an expression with an integer-type value.
3. *count* is an expression with an integer-type value.

Delete(dest, index, count) removes *count* characters from the value of *dest*, beginning at *dest[index]*. If $index < 1$ and/or $index + count > length(source)$, i.e. if character positions outside the range $1..length(source)$ are implicitly referenced, it is not an error. However, only the characters that lie within that range are deleted.

10.5.6 The Omit Function

Deletes a substring of specified length from a specified position within a string value and returns the result.

Result Type: *string-type*

Parameter List: **`omit(str, index, count)`**

1. *str* is a value of string-type.
2. *index* is an expression with an integer-type value.
3. *count* is an expression with an integer-type value.

Omit(str, index, count) removes *count* characters from the value of *str*, beginning at *dest[index]*, and returns the resulting string value. This is similar to *delete* except that *str* is not affected; the resulting string value is returned as the value of the function instead.

10.5.7 The Insert Procedure

Inserts a substring into the value of a string variable, at a specified position.

Parameter List: **`insert(source, dest, index)`**

1. *source* is an expression with a value of string-type.

2. *dest* is a variable-reference that refers to a variable of string-type. This is a variable parameter.
3. *index* is an expression with an integer-type value.

Insert(source, dest, index) inserts *source* into *dest*. The first character of *source* becomes *dest[index]*. If *index* < 1 or *index* > *length(dest)*, it is not an error. If *index* < 1 then *source* is appended to the left of *dest*. If *index* > *length(dest)* then *source* is appended to the right of *dest*. It is an error, however, if the length of the resulting string is greater than 255.

10.5.8 The Include Function

Inserts a substring into a string value, at a specified position, and returns the result.

Result Type: string-type

Parameter List: include(source, str, index)

1. *source* is an expression with a value of string-type.
2. *str* is a value of string-type.
3. *index* is an expression with an integer-type value.

Include(source, str, index) inserts *source* into the value of *str* at *str[index]* and returns the result. This is similar to *insert* except that *str* is not affected; the resulting string value is returned as the value of the function instead.

10.6 Lightspeed Pascal Window Manipulation Procedures

10.6.1 The HideAll Procedure

Parameter List: HideAll

HideAll causes all of the windows on the Lightspeed Pascal desktop to be hidden. All of these windows may be revealed again via the **Windows** menu. In addition, the Text and Drawing windows may be revealed by calling the procedures described below.

10.6.2 The ShowText Procedure

Parameter List: ShowText

ShowText causes the Text window to be revealed and to become the active window. The size and position of the window is unchanged.

10.6.3 The ShowDrawing Procedure

Parameter List: **ShowDrawing**

ShowDrawing causes the Drawing window to be revealed and to become the active window. The size and position of the window is unchanged.

10.6.4 The SetTextRect Procedure

Parameter List: **SetTextRect (WindowRect)**

1. *WindowRect* is a value of type *Rect* (see Appendix C).

WindowRect is a rectangle in QuickDraw's global coordinate system (see Appendix C) that determines the position and size of the Text window on the Macintosh screen.

10.6.5 The SetDrawingRect Procedure

Parameter List: **SetDrawingRect (WindowRect)**

1. *WindowRect* is a value of type *Rect* (see Appendix C).

WindowRect is a rectangle in QuickDraw's global coordinate system (see Appendix C) that determines the position and size of the Drawing window on the Macintosh screen.

10.6.6 The GetTextRect Procedure

Parameter List: **GetTextRect (WindowRect)**

1. *WindowRect* is a variable-reference of type *Rect* (see Appendix C).

GetTextRect returns a rectangle in *WindowRect* with coordinates in QuickDraw's global coordinate system (see Appendix C). This rectangle indicates the current size and position of the Text window.

10.6.7 The GetDrawingRect Procedure

Parameter List: **GetDrawingRect (WindowRect)**

1. *WindowRect* is a variable-reference of type *Rect* (see Appendix C).

GetDrawingRect returns a rectangle in *WindowRect* with coordinates in QuickDraw's global coordinate system (see Appendix C). This rectangle indicates the current size and position of the Drawing window.

10.6.8 The SaveDrawing Procedure

Parameter List: **SaveDrawing(title)**

1. *title* is a string-type value that must contain a valid file name for a file-structured device (see Section 9.5).

SaveDrawing saves the contents of the Drawing window as a *picture file* that may be read by MacPaint. The *title* string contains the name of the picture file to be created. If a file by that name already exists, it is overwritten.

Warning: *SaveDrawing* actually saves the contents of the current QuickDraw GrafPort (see Appendix C). However, unless you specifically change the current port to be another port, the current port will always be the Drawing window's GrafPort when your program is running.

10.7 Miscellaneous Procedures and Functions

10.7.1 The Sizeof Function

Returns the number of bytes occupied by a specified variable, or by any variable of a specified type.

Result Type: *integer*

Parameter List: **sizeof(id)**

1. *id* is either a variable-identifier or a type-identifier.

Sizeof(id) returns the number of bytes of memory occupied by *id*. *Id* cannot have any modifier (e.g. *aRec.field*). If *id* is a variable-identifier; if *id* is a type-identifier, it returns the number of bytes occupied by any variable of type *id*.

10.7.2 Logical Operations

Result Type: *integer* or *longint*

Parameter List: **BitAnd(aNum1, aNum2)**
BitOr(aNum1, aNum2)
BitXor(aNum1, aNum2)
BitNot(aNum)

1. *aNum*, *aNum1*, and *aNum2* are either *integer* or *longint* values.

BitAnd returns the logical *and* of the bits of the two given values.

BitOr returns the logical *or* of the bits of the two given values.

BitXor returns the logical *xor* of the bits of the two given values.

BitNot returns the logical *not* of the bits of the given value.

If the operands *BitAnd*, *BitOr*, *BitXor*, or *BitNot* routines are of the integer-type *integer* the result is always of type *integer*. If one or both operands *BitAnd*, *BitOr*, *BitXor*, or *BitNot* routines are of the integer-type *longint*, the result is always of type *longint*.

Note: These routines generate inline code, as opposed to Toolbox traps. The results will be identical to the Toolbox trap results.

10.7.3 Operations on Longint Values

Result Type: *integer*

Parameter List: **HiWord(long)**
 LoWord(long)

1. *long* is a *longint* value.

HiWord and *LoWord* return the upper and lower 16-bits of a *longint* value respectively as an *integer* value.

Note: These routines generate inline code, as opposed to Toolbox traps. The results will be identical to the Toolbox trap results.

10.7.4 The WriteDraw Procedure

WriteDraw is similar to *write*, except that the text output goes to the current GrafPort at current PenLoc instead of to a textfile or to the Text window.

Parameter List: **WriteDraw(*p*₁ [, *p*₂, ..., *p*_{*n*}])**

1. *p*₁, ..., *p*_{*n*} are *write-parameters*. Each *write-parameter* includes an *output expression*, whose value is to be written to the file. As explained in Section 9.4.3.1, a *write-parameter* may also contain the specifications of a field-width and a number of decimal places. Each *output expression* must have a result of char-type, an integer-type, a real-type, a string-type, a packed-string-type, or an enumerated-type. At least one *write-parameter* must be present.

WriteDraw takes the same parameter list as *write* (see Section 9.4.3), except that no file parameter is ever given. The text that results from the evaluation of each *write-parameter* is written in the current GrafPort starting at the current pen position (see Appendix C).

Do not forget to set current PenLoc before using *WriteDraw*.

10.7.5 The StringOf Function

StringOf is also similar to *write*, except that the text output is returned as a string-type value instead of being written to a textfile or to the Text window.

Result Type: *string-type*

Parameter List: **StringOf**(*p*₁ [, *p*₂ , ..., *p*_{*n*}])

1. *p*₁ , ..., *p*_{*n*} are *write-parameters*. Each write-parameter includes an *output expression*, whose value is to be written to the file. As explained in Section 9.4.3.1, a write-parameter may also contain the specifications of a field-width and a number of decimal places. Each output expression must have a result of char-type, an integer-type, a real-type, a string-type, a packed-string-type, or an enumerated-type. At least one write-parameter must be present.

StringOf takes the same parameter list as *write* (see Section 9.4.3), except that no file parameter is ever given. The text that results from the evaluation of each write-parameter is accumulated as a string-type value that is the result of the function call.

10.7.6 The ReadString Procedure

ReadString is similar to *read*, except that the text is read from a string parameter instead of a textfile.

Parameter List: **ReadString**(*s*, *v*₁ [, *v*₂ , ..., *v*_{*n*}])

1. *s* is a string-type value.
2. Each *v* is a variable-reference that refers to a variable of one of the following types:
 - *Char* or a subrange thereof.
 - An integer-type: *integer* (or a subrange thereof) or *longint*.
 - A real-type: *real*, *double*, *extended*, or *computational*.
 - An enumerated-type (including *boolean*) or a subrange thereof.
 - A string-type.

ReadString reads text from the string value *s* just as if it were doing a *read* (see Section 9.4.1) from a textfile. The values read are placed in the *v* parameters in the given order. Just as it is an error to attempt to read beyond the end of a file, it is an error to attempt to read characters beyond the end of the string value *s*.

10.7.7 The OldFileName Function

Returns the title of an existing disk file selected by the user.

Result Type: string-type

Parameter List: OldFileName(Prompt)

1. *prompt* is a string-type value.

OldFileName causes a dialog box to appear on the Macintosh screen. The *Prompt* string is displayed to give the user some indication of what the box is asking for. With the dialog box, the user can peruse the existing files on any number of disks and select one of them. *OldFileName* returns a string value that is the title of the file the user selected, which can in turn be given to *reset*, *rewrite*, or *open*.

10.7.8 The NewFileName Function

Returns the title of a new disk file selected by the user.

Result Type: string-type

Parameter List: NewFileName(Prompt [, Default Name])

1. *Prompt* is a string-type value.

NewFileName causes a dialog box to appear on the Macintosh screen. The *Prompt* string is displayed within this box to give the user some indication of what the box is asking for. With the dialog box, the user can select any disk and enter the name (or choose the default name) of a file to be created on that disk. *NewFileName* returns a string value that is the title of the file the user selected, which can in turn be given to *reset*, *rewrite*, or *open*.

The *Default Name* string is displayed in the dialog box as selected text, and will be the string returned unless the user enters another name.

10.7.9 The Synch Procedure

Parameter List: Synch

The *synch* procedure is used to synchronize your program's actions with the Macintosh screen's drawing cycle, which occurs every 60th of a second. This may, in particular, be used to synchronize calls to QuickDraw (see Appendix C) with the screen drawing cycle to avoid unnecessary flicker and scanning bar phenomena when moving things around quickly in the Drawing window.

When you call the *synch* procedure, the procedure will not return until the screen has reached the point in its cycle where the electron beam has returned to the top of the screen and is about to redraw the entire picture. Depending on where your QuickDraw calls are going to be drawing on the screen, additional delays may be necessary to synchronize the drawing with the timing of the electron beam's scanning of the screen.

10.7.10 The Note Procedure

Parameter List: **Note(Frequency, Amplitude, Duration)**

1. *Frequency* is an *longint* value in the range 12..783360.
2. *Amplitude* is an integer-type value in the range 0..255.
3. *Duration* is an integer-type value in the range 0..255.

Note causes a single square-wave tone to be generated, of the given amplitude, duration, and frequency (the frequency is specified in hertz).

10.7.11 The InLine Procedures

Parameter List: **InlineP(Trap [, p₁, p₂, ..., p_n])**
BInlineF(Trap [, p₁, p₂, ..., p_n])
WInlineF(Trap [, p₁, p₂, ..., p_n])
LInlineF(Trap [, p₁, p₂, ..., p_n])

1. *Trap* is an expression with an integer value that indicates the number of the trap to be called.
2. Each *p* (if given) is an expression with any type value.

The InLine procedures provides the ability to call the stack-based Macintosh Toolbox routines. The InLine procedures work by disabling all type and parameter checking normally used in Lightspeed Pascal. This makes it possible for just four predefined routines to invoke any stack-based Toolbox trap.

Toolbox routines can be either *functions* or *procedures*. Functions can return either a *byte* (8-bits), a *word* (16-bits), or a *longword* 32-bits as results.

Toolbox *functions* can, depending on the result type, be invoked with one of the three following routines:

- BinlineF - Toolbox functions which return a *byte* (*boolean*)
- WInlineF - Toolbox functions which return a *word* (*integer*)
- LInlineF - Toolbox functions which return a *longword* (*longint*)

Toolbox *procedures* can be invoked with the following routine:

- InlineP

All procedures and functions use the same basic parameter-passing mechanism. Parameters are passed either by reference (variable parameters) or by value. To force a parameter to an InLine routine to be passed by reference, the @ operator may be applied to the name of the variable that is to be passed. For example,

@WindPtr

will force a reference to *WindPtr* to be passed to a Toolbox routine. Simply using a variable, constant, or literal value will pass a parameter by value.

The first parameter to any InLine call is the value which specifies the trap number. The trap number indicates which Toolbox routine is to be called. All subsequent parameters must exactly match the number and type of the parameters for the particular Toolbox routine being called.

Warning: Because there is no type checking, none of Lightspeed Pascal's usual implicit type-coercion will be performed (e.g., integer to longint). All parameters **MUST** match exactly the parameters for the particular Toolbox routine being called.

Note: The InLine procedures are provided only for compatibility with Macintosh Pascal. Lightspeed Pascal provides direct access to the Macintosh toolbox and therefore the only need for InLine is for porting Macintosh Pascal programs which used these routines.

10.7.12 The Generic Procedure

Parameter List: **Generic (InstructionWord, Registers)**

1. *InstructionWord* is an expression with an integer value that indicates the instruction to be executed.
2. *Registers* is a variable-reference of type *RegisterRecord* (see below) that indicates the values to be written to the MC68000 registers.

Generic permits you to call register-based Macintosh ROM routines. It can also be used to execute any machine-language code that you have stored in a Pascal data structure.

RegisterRecord denotes a data structure consisting of 13 32-bit values - five address register values (A0..A4), followed by eight data register values (D0..D7). The exact type of this structure is immaterial. For example, you could declare:

```
var registers: record
    a: array[0..4] of longint;
    d: array[0..7] of longint
end;
```

The register values passed to *Generic* are written to the MC68000 registers. Then the one-word instruction denoted by the *InstructionWord* argument is executed. Finally, the *Registers* structure is updated with the (possibly) new values of the MC68000 registers before *Generic* returns to the program.

Generic disallows stack operations it seems!

Usually, *Generic* will be used to execute a register-based Toolbox trap. In such cases, the value you pass to *Generic* via the *InstructionWord* argument is the trap value.

The *InstructionWord* argument to *Generic* does not have to be a trap value, it can be any 16-bit MC68000 instruction.

Note: The *Generic* procedure is provided only for compatibility with Macintosh Pascal. Lightspeed Pascal provides direct access to the Macintosh toolbox as well as the ability to call assembly language routines. Therefore, the only need for *Generic* is for porting Macintosh Pascal programs which used this routine.

10.7.13 The Macsbug Support Procedures

Parameter list: *Debugger*
DebugStr(message)

1. *Message* is a string-type value.

Debugger and *DebugStr* cause a User Break which interrupts the execution of the program and transfers control to the Macsbug low-level debugger. If *DebugStr* is used, the *message* argument is displayed.

If Macsbug is not installed, then calls to *Debugger* and *DebugStr* give the run time error message "Macsbug not installed."

Refer to Appendix G for more information on using Macsbug.

PART THREE

APPENDICES

Appendix A

ANS Pascal Compatibility

Introduction	A-3
Exceptions to ANS Pascal Requirement	A-3
Extensions to ANS Pascal	A-4
Implementation-Dependent Features	A-6
Treatment of Errors	A-6

ANS Pascal Compatibility

Introduction

This appendix describes the relationship between Lightspeed Pascal and the requirements of ANSI/IEEE770X3.97-1983, American National Standard Pascal (ANS Pascal).

Exceptions to ANS Pascal Requirements

Lightspeed Pascal complies with the requirements of ANSI/IEEE770X3.97-1983 with the following exceptions:

- In ANS Pascal, the special-symbol `@` is an alternative representation for the special-symbol `^`, and is required to be treated identically to `^` wherever it appears. In Lightspeed Pascal, the special-symbol `@` is an operator and is never treated identically to `^`.
- In ANS Pascal, identifiers may be of any length and all characters are significant. In Lightspeed Pascal, all characters in identifiers are significant, but the largest identifier is restricted to 255 characters.
- In ANS Pascal, a character-string of length 1 is a char-type value and a character-string of length *n* is a value of a packed-string-type (a ***packed array*** *[1..n] of char* -- referred to as a *string-type* in ANS Pascal) with *n* components. In Lightspeed Pascal, all quoted character-strings are string-type values. However, the compatibility and assignment-compatibility rules in Lightspeed Pascal make its behavior with respect to character-strings compatible with ANS Pascal.
- In ANS Pascal, all values of a tag type must appear once for a given variant part. In Lightspeed Pascal, this requirement is not enforced.
- In ANS Pascal, a function block must contain at least one assignment statement assigning a value to the function identifier. In Lightspeed Pascal, this requirement is not enforced.
- In ANS Pascal, a field that is the selector of a variant part of a record may not be an actual variable parameter. In Lightspeed Pascal, this requirement is not enforced.
- In ANS Pascal, no statements that *threaten* the value of a control-variable of a for-statement are allowed. In Lightspeed Pascal, this requirement is not enforced. Changing the value of the control variable produces indeterminate results.

- In Lightspeed Pascal, only the standard file variables `input` and `output` are allowed as program parameters.

Note: There is no automatic means, in Lightspeed Pascal, of determining whether or not a program violates any of the exceptions listed above.

Extensions to ANS Pascal

The following Lightspeed Pascal features are extensions to Pascal as specified by ANSI/IEEE770X3.97-1983:

- The following are word-symbols in Lightspeed Pascal:

<i>inline</i>	<i>interface</i>	<i>implementation</i>	<i>otherwise</i>
<i>string</i>	<i>uses</i>	<i>unit</i>	

- An identifier may have an underscore appearing anywhere following the initial letter of the identifier.
- Lightspeed Pascal supports relaxation of the ordering of declarations. There may be any number of declaration parts in any order.
- Lightspeed Pascal supports the additional integer-type *longint* and the additional real-types *double*, *computational*, and *extended*.
- A signed constant-identifier may denote a value of type *integer*, *longint*, or *extended*.
- In Lightspeed Pascal, the result of arithmetic performed on *integer* operands is *integer*. The result of arithmetic performed on *longint* operands is *longint*. All mixed *integer* and *longint* operands are converted to *longint* before arithmetic is performed, and the result is *longint*. A *longint* value may be used wherever an *integer* value is required if the value falls in the range *-maxint..maxint*.
- All integer-type and real-type operands are converted to *extended* before any real arithmetic is performed, and the result is always *extended*. An *extended* value may be used wherever a *real*, *double*, or *computational* value is required, provided the value falls within the range of values permissible.
- Lightspeed Pascal supports string-types, which are compatible with other string-types, packed-string-types, and char-type.
- In Lightspeed Pascal, the assignment-compatibility rules have been extended to allow the mixing of string-types, packed-string-types, and char-type where appropriate.
- The result-type of a function is not restricted to simple and pointer-types; functions may return values of any type.
- Individual char-type components of string-type variables may be referenced as though the string were a one-dimensional array.

- String-types may be compared with char-type and packed-string-type values.
- The *@* operator is provided for obtaining the address of a variable, procedure, or function.
- Lightspeed Pascal has an optional ***otherwise*** clause for the case-statement.
- Lightspeed Pascal supports the use of the *indefinite-string-type*, i.e. the word-symbol ***string***, as a value or variable-parameter type.
- An optional second parameter may be given to *reset* and *rewrite* to associate a file variable with an external file.
- Instead of *reset* or *rewrite*, a file may be opened with *open* to allow random read/write access to a file.
- An explicit *close* procedure is supplied for those file variables associated with external files.
- A *seek* procedure may be used for random-access to file components.
- A *filepos* function returns the component number of the current file position, a value that may be used in subsequent *seek*'s.
- String-type and enumerated-type values may be read from textfiles with *read* and *readln*.
- String-type and enumerated-type values may be written to textfiles with *write* and *writeln*.
- "Lazy I/O" is used to permit interactive and non-interactive I/O to be treated identically.
- In Lightspeed Pascal, the *ord* function may be applied to a pointer-type value, facilitating address arithmetic.
- The *ord4* function is provided in Lightspeed Pascal for converting an ordinal-type or pointer-type value to a *longint*.
- The *pointer* function is provided in Lightspeed Pascal for converting an integer-type value to a pointer-type value.
- Lightspeed Pascal includes a set of string procedures and functions.
- The *sizeof* function is provided for obtaining the storage size of a variable or type.
- ***Inline*** routines are supported for imbedding machine code in a program.
- ***Units*** are supported for modular construction of programs and separate compilation.
- Lightspeed Pascal supports type-casting of values.

- Lightspeed Pascal supports all of the Macintosh Toolbox/OS constants, types, variables, procedures, and functions as predefined identifiers.

Note: There is no automatic means, in Lightspeed Pascal, of distinguishing between a program that uses extensions and one that does not.

Implementation-Dependent Features

The effect of using an implementation-dependent feature of Pascal, as defined by ANSI/IEEE770X3.97-1983, is unspecified as described in the preface to the Language Reference.

Treatment of Errors

This section defines those errors listed in Appendix D of the ANS Pascal standard that are not automatically detected and reported by Lightspeed Pascal. The number of each error listed below is the number under which the error is listed in the standard's appendix. The wording of the description of the error, however, differs from the wording in the standard.

2. If t is the tag-field of a variant-part, and if f is a field within the currently active variant of that variant-part, then it is an error to attempt to alter the value of t while a reference to f exists.
4. If p is a pointer-type value, it is an error to reference p^{\wedge} if the value of p is undefined.
5. If p is a pointer-type value, it is an error to attempt to *dispose* of p while a reference to p^{\wedge} exists.
6. If f is a file-type variable, it is an error to attempt to close f or alter the current file position of f while a reference to f^{\wedge} exists.
19. If a pointer-type variable p is assigned a value by $new(p, c_1, c_2, \dots, c_n)$, it is an error to attempt to make any other variants than those selected by the case-constants in *new* become the active variant.
20. If a pointer-type variable p is assigned a value by $new(p, c_1, c_2, \dots, c_n)$, it is an error to attempt to *dispose* of p without supplying the same list of case-constants in the same order.
21. Same as 20.
22. Same as 20.
24. It is an error to attempt to *dispose* of p if the value of p is undefined.
25. If a pointer-type variable p is assigned a value by $new(p, c_1, c_2, \dots, c_n)$, it is an error to reference the entire variable p^{\wedge} in an expression, as an actual variable-parameter, or as the destination of an assignment-statement.

- 27. For $pack(a, i, z)$, it is an error if any of the referenced components of a have undefined values.
- 30. For $unpack(z, a, i)$, it is an error if any of the components of z have undefined values.
- 43. It is an error to reference a variable in an expression if the value of that variable is undefined.
- 48. It is an error if the result of the activation of a function is undefined when that activation is complete.
- 51. It is an error if none of the case constants is equal to the value of the case index upon entry to a case statement.

Appendix B

Porting to Lightspeed Pascal

Introduction	B-3
Program Size	B-3
Identifier Length	B-4
Reserved words	B-4
Comments and Directives	B-4
Types	B-5
Data Representation	B-6
Data Initialization	B-6
Operators	B-7
Integer Arithmetic	B-7
Program Parameters	B-8
Predefined Procedures and Functions	B-8
Standard Units	B-8
Input/Output	B-9
Segmentation	B-10
Run Time Environment	B-10
Extensions	B-10

Porting to Lightspeed Pascal

Introduction

The Pascal language was designed by Professor Niklaus Wirth primarily as a tool for teaching systematic programming. It has evolved into one of the most popular programming languages, the main attraction being its emphasis on rigid structure and strong typing. In the evolution process most Pascal implementations have added a number of language extensions. Since most such extensions are unique to a particular implementation, some work is required to port Pascal programs from one computer to another.

Different implementations have interpreted the semantics of the unadorned language in different ways. This issue was addressed by the Joint ANSI/X3J9-IEEE Pascal Standards Committee, which in 1982 approved a standard language (informally called ANS Pascal) to promote the portability of Pascal programs between implementations.

Lightspeed Pascal is intended to conform, as closely as possible, to ANS Pascal (see Appendix A), while providing extensions necessary for the development of serious (Macintosh) applications. We have tried to keep such extensions to a minimum, choosing features common to many Pascal implementations, while retaining compatibility with existing systems, notably Macintosh Pascal.

This appendix is designed to aid you in porting existing programs to Lightspeed Pascal. It provides a comprehensive list of areas in which Lightspeed Pascal may differ from other implementations, along with suggestions for modifying programs to run under Lightspeed Pascal. Wherever appropriate, specific mention is made of other Pascal systems available for the Macintosh, specifically Macintosh Pascal (THINK Technologies and Apple Computer), Lisa Pascal (Apple Computer), and MacLanguage Series Pascal (TML Systems).

There are probably only a few places where your programs will actually require modification. Your efforts will be rewarded almost immediately with the increased productivity resulting from the benefits of the Lightspeed Pascal environment.

Program Size

Lightspeed Pascal limits the size of any individual source file to around 2000-3000 lines depending on the line density. This limitation provides improved edit and compile performance, and relates well to inherent limitations such as segment size. However, programs developed using other compilers may consist of larger source files, particularly those developed without the benefit of separate compilation.

If you attempt to load a file that exceeds this limitation, a warning is given that some lines have been lost. The best approach is to break the file up into smaller units, using an editor capable of dealing with large files.

If a large file contains procedures and functions which are to be allocated in different segments (as is typical with Lisa Pascal programs) you should break up the file based on its required segmentation; see the section on Segmentation below.

Identifier Length

Most Pascal compilers only check the spelling of identifiers up to 8 or 16 characters. In Lightspeed Pascal, all characters in an identifier are significant, up to the maximum length (255). For example:

```
var
    aVeryVeryLongName: integer;
begin
    aVeryVeryLongNme := 0;
end;
```

This is accepted by many compilers because the difference between the two identifiers occurs beyond the last significant character. Lightspeed Pascal reports the misspelling as an undeclared identifier.

Reserved Words

Lightspeed Pascal has added the following word-symbols which may not be used as identifiers:

implementation	inline	interface	otherwise
string	uses	unit	

If you attempt to compile a program which uses one of these as an identifier, Lightspeed Pascal will report an error; this type of error is also detected by the editor, and will appear in outline form to indicate that it is syntactically invalid.

Comments and Directives

Some Pascal compilers treat the comment delimiter pairs { } and (* *) differently, allowing comments which use one delimiter to be "commented out" using the other delimiter. For example, in Lisa Pascal the code sequence

```
if ResError <> noErr then {report resource error}
    alertStatus := Alert (MyRsrcAlert, nil);
```

could be commented out in the following fashion:

```
(* if ResError <> noErr then {report resource error}
    alertStatus := Alert (MyRsrcAlert, nil);
*)
```

In Lightspeed Pascal, the different comment delimiter pairs are always treated the same, so the above technique cannot be used (comments do not nest). Furthermore, comments may not cross line boundaries. Lightspeed Pascal attempts to convert multi-line comments into a number of single line comments. However, nested comments may not be converted properly.

Most Pascal implementations support special *compiler directive* comments. These comments are typically of the form `{ $x+ }` or `{ $x- }`, where *x* is a one or two character mnemonic for the particular option. Lightspeed Pascal supports a number of compiler directives; these are described in detail in Chapter 13 of the User's Guide.

Compiler directives are not addressed in the ANS Pascal standard, and thus vary greatly between implementations. For example, the Lisa Pascal compiler supports

<code>{ \$I filename }</code>	include <i>filename</i> in text
<code>{ \$S segmentname }</code>	place code in segment <i>segmentname</i>
<code>{ \$OV+ }</code>	turn on overflow checking
<code>{ \$R- }</code>	turn off range checking

and a number of others, most of which have no analog in Lightspeed Pascal. It is best to assume that all compiler directives are nonportable, and should be changed or removed.

It should be noted that most Lightspeed Pascal compiler directives may be enabled and disabled via the Project window; this is both more convenient and more portable.

Types

Different implementations of Pascal place different limitations on predefined and user defined data types. These limitations typically reflect the chosen representation of data, or constraints imposed by the underlying hardware architecture. For example, the ANS standard requires that the type *integer* contain all values in the range *-maxint* to *maxint*, but accepts the fact that these values are machine dependent and may differ across implementations.

Lightspeed Pascal imposes the following data type limitations:

- Integers are limited to the range -32767 to 32767.
- Integer subranges are also limited to the range -32767 to 32767, i.e. no *longint* subranges are allowed.
- Enumerated types may contain no more than 256 distinct values.
- Structured types may occupy at most 32766 bytes.

There are also areas in which Lightspeed Pascal exceeds limitations imposed by other Pascal implementations. For example:

- Set types up to the full *set of integer* are supported, including negative-value elements.

- Floating-point values of *single*, *double*, *computational* and *extended* precision are supported.
- Complete compatibility exists between string-types, packed-string-types, and char-type.

For a complete description of all Lightspeed Pascal data types, refer to Section 3 of the Language Reference.

Data Representation

The area likely to have the greatest diversity among Pascal implementations is the representation of data; this is the area most subject to hardware differences and personal judgement. Lightspeed Pascal follows the Lisa Pascal (and thus Macintosh Toolbox) conventions for ordinal types. For example, the *integer* type is implemented as a 16 bit value to take advantage of the more efficient 16 bit instructions of the 68000 processor. The 32 bit integer type *longint* is provided for dealing with values which exceed the range of 16 bit integers.

The data type *char* is notable because it is also implemented as a 16 bit (rather than the more natural 8 bit) value; it occupies 8 bits only when packed. The ramifications of this are subtle. The predefined type *text* is usually assumed to have the same representation as ***file of char***; Macintosh Pascal assumes that this is the case. In Lightspeed Pascal, *text* has the same representation as ***packed file of char***. This may cause difficulty when porting programs which use ***file of char***.

The Lisa Pascal compiler provides a more comprehensive packing algorithm than Lightspeed Pascal; for example, the type definition

```
type
    KeyMap = packed array [0..127] of boolean;
```

causes each element of a *KeyMap* to occupy one bit. In Lightspeed Pascal, bit-packing is never performed, and this type is implemented differently:

```
type
    KeyMap = array [0..3] of longint;
```

Programs which depend upon bit-packing of data are not portable to Lightspeed Pascal without modification.

The representation of data in Lightspeed Pascal is described more completely in Chapter 11 of the User's Guide.

Data Initialization

Certain Pascal implementations guarantee specific values for uninitialized variables. For example, Macintosh Pascal initializes all global and local variables (and function results) to 0. In Lightspeed Pascal, no variables are initialized. Programs which (perhaps inadvertently) depend on variables being initialized may result in unusual run-time behavior.

Operators

The implementation of operators in Lightspeed Pascal conforms for the most part to the ANSI standard, differing only in areas where Lightspeed Pascal provides additional data types (e.g. **string**, *longint*, *extended*). Porting difficulties may occur in expressions involving integer arithmetic overflow (see the following section).

Lisa Pascal programs which use the **mod** operator may yield different results when ported (if the dividend is negative). Lightspeed Pascal (correctly) implements **mod** according to the rules for modulo arithmetic, while Lisa Pascal implements it as a simple remainder operator.

Many compilers support an exponentiation operator ****** for integer or floating-point operands. In Lightspeed Pascal, floating-point exponentiation can be easily implemented, based on the following identity:

$$x ** y = \text{Exp} (\text{Ln} (x) * y)$$

If you port a program which uses integer exponentiation, you may have to write a simple exponentiation function.

Integer Arithmetic

Lightspeed Pascal performs all integer arithmetic in either 16 bits or 32 bits depending on the type of the operands. For binary operators, if both operands are 16 bits (or smaller) the operation is performed in 16 bits; if either or both operands is 32 bits, the operation is performed (less efficiently) in 32 bits.

The ramifications of this strategy are illustrated by the following example:

```
var
    i, j: integer;
begin
    i := 20000;
    j := 30000;
    writeln (i+j);
end;
```

This program fragment will cause an overflow error in Lightspeed Pascal, because the sum 20000+30000 is too large to be represented in 16 bits. Other implementations (notably Macintosh Pascal) may perform the operation in 32 bits, avoiding the overflow at the expense of execution efficiency. You can force the arithmetic to be performed in 32 bits by casting the operands to *longint* or by using the *Ord4* function.

A similar problem arises in programs ported from Macintosh Pascal which use inline procedures and functions (*InlineP*, *BInlineF*, *WInlineF*, *LInlineF*). The statement

```
InlineP ($A9D1, 0+0, 65535, hTE);
```

attempts to call *TESetSelect* with a (longword) 0 for *selStart*. In Lightspeed Pascal, this results in a word 0 being pushed instead, misaligning the stack and (probably) resulting in a fatal error. You can cast the *integer* value to a *longint*, but statements like that above are better replaced by a call to the equivalent Macintosh Toolbox routine:

```
TESetSelect (0, 65535, hTE);
```

This will perform the proper argument type checking and conversion, and is both easier to understand and more efficient.

Program Parameters

Lightspeed Pascal only allows the standard file-variables *input* and *output* as program-parameters. Occurrences of other file-variables in a program heading are reported as errors. However, these can be removed without affecting the semantics of the program.

Predefined Procedures and Functions

Lightspeed Pascal, like Macintosh Pascal, provides a large number of predefined procedures and functions in addition to those required by the standard. Routines have been added to facilitate address arithmetic, string handling, I/O, and other common tasks. On top of this, nearly all of the procedures and functions (and constants and types) described in *Inside Macintosh* are supplied as predefines (some are supplied as units: see below).

Different implementations of Pascal are likely to provide different predefined procedures and functions, many of which have functional analogs in Lightspeed Pascal. Programs which depend upon nonstandard predefines (most programs fall into this category, for reasons described in the introduction) will require modification. For example, the following Lisa Pascal predefines are not supported:

<i>exit</i>	<i>halt</i>	<i>heapresult</i>	<i>mark</i>
<i>release</i>	<i>memavail</i>	<i>moveleft</i>	<i>moveright</i>
<i>scaneq</i>	<i>scanne</i>	<i>fillchar</i>	
<i>band</i>	<i>bor</i>	<i>bxor</i>	<i>bnot</i>
<i>bsl</i>	<i>bsr</i>	<i>brotl</i>	<i>brotr</i>
<i>btst</i>	<i>bclr</i>	<i>bset</i>	

For a complete description of the Lightspeed Pascal predefines, see Section 10 of the Language Reference.

Standard Units

Pascal compilers for the Macintosh typically provide a standard set of units (Lisa Pascal), libraries (Macintosh Pascal) or include files (MacLanguage Series Pascal) which define some or all of the constants, types, procedures, etc., described in *Inside Macintosh*, and whose declarations can be made available to any program or unit. For example, most Lisa Pascal programs begin with the following uses-clause:

```
uses
    MemTypes, QuickDraw, OSIntf, ToolIntf, PackIntf;
```

Similarly, Macintosh Pascal programs often start with:

uses

QuickDraw1, QuickDraw2;

In Lightspeed Pascal, most of the declarations in Inside Macintosh are predefined, and these uses-clauses are not required (or allowed).

Certain of the Inside Macintosh declarations, such as the AppleTalk Manager, are supplied as libraries. Each is supplied as two files: a source (interface) file, and a library (code) file. Both of these files must be added to the project. The interface to the library is in the form of a *unit*. The unit name must appear in a uses-clause in the file that accesses the library. Methods for using these libraries are described in Chapter 8 of the User's Guide. The exact syntax for units is described in Section 8.3 of the Language Reference. The interfaces are documented in Appendix E.

Input/Output

This is an area in which many Pascal implementations diverge, particularly in their treatment of interactive I/O. Lightspeed Pascal supports I/O with ANS Pascal semantics and the following extensions:

- An optional second parameter may be given to *reset* and *rewrite* to associate a file variable with an external file. A file opened *reset* or *rewrite* is read-only or write-only respectively, and in both cases the file may only be accessed sequentially.
- Instead of *reset* and *rewrite*, a file may be opened with *open* to allow random read/write access.
- A *close* procedure is supplied for those file variables associated with external files. There are no optional *close* parameters (unlike Lisa Pascal, for example).
- A *seek* procedure may be used for random-access to file components.
- A *filepos* function returns the component number of the current file position, a value that may be used in a subsequent *seek*.
- String-type and enumerated-type values may be read from textfiles using *read* and *readln*.
- String-type and enumerated-type values may be written to textfiles using *write* and *writeln*.
- Lazy I/O is used to permit both interactive and non-interactive I/O to be handled identically.

Lightspeed Pascal is identical to Macintosh Pascal in its treatment of I/O. Other compilers are likely to have different extensions. Programs which read and write to the predefined file variables *input* and *output* interactively may cause some difficulty; see Section 9 of the Language Reference for a complete explanation of Lazy I/O and its ramifications.

Segmentation

Macintosh programs are usually composed of a number of segments, each of which can contain up to 32K of code. Segmentation allows programs to be written which are too large to fit in memory; segments can be loaded and unloaded as needed.

Most Pascal implementations for the Macintosh support segmentation. In Lisa Pascal and MacLanguage Series Pascal, segmentation is accomplished (on a procedure by procedure basis) via a compiler option:

```
{$S PrintSeg }  
procedure PrintFile (whichFile: integer);  
begin  
  ...  
end;  
  
{$S FileSeg }  
procedure CloseFile (whichFile: integer);  
begin  
  ...  
end;
```

The above directives place the code for procedure *PrintFile* in segment *PrintSeg*, and the code for procedure *CloseFile* in segment *FileSeg*.

In Lightspeed Pascal, programs are divided into segments on a file by file basis. The Project Window, when viewed by segment, shows which files are in which segments, and provides a visual metaphor (dragging) for moving files between segments. If you are porting a program in which a single file contributes to multiple segments, some reorganization will be necessary.

Run Time Environment

Macintosh programs usually begin with a sequence of initialization calls to various portions of the Macintosh Toolbox. In Lightspeed Pascal, the following initializations are performed automatically for your application:

```
InitGraf (@thePort);  
InitFonts;  
InitWindows;  
InitMenus;  
TEInit;  
InitDialogs (nil);  
InitCursor;  
SetApplLimit (<current value of A7> - <Run Options stack size>)  
MaxApplZone;
```

Although it is difficult to imagine a program which would not require these calls, you can disable their automatic execution using the `{$I-}` directive (see Chapter 13 of the User's Guide).

Extensions

If you are porting Lisa Pascal programs, you may run into difficulty if the programs use some of the Lisa Pascal extensions. The following features are not supported by Lightspeed Pascal:

- *Cycle* and *Leave* Statements. These are easily replaced by an equivalent **goto** statement.
- Constant Expressions. In Lisa Pascal, the following constant declarations are accepted:

```
const
    limit = maxSize - 1;
    CR = Chr ($OD);
```

In Lightspeed Pascal, constant expressions are not allowed.

- Case List Subranges. In Lisa Pascal, constant lists in **case** statements may range over a set of values without explicitly listing them all, i.e.

```
case today of
    monday..friday: ...
    saturday: ...
    sunday: ...
end;
```

In Lightspeed Pascal, the subranges must be listed explicitly:

```
case today of
    monday, tuesday, wednesday, thursday, friday: ...
    saturday: ...
    sunday: ...
end;
```

- Generalized Function Results. In Lisa Pascal, as in Lightspeed Pascal, functions may return values of any type. This feature has been generalized in Lisa Pascal so that function calls may be treated as simple variables so that they may be indexed, dereferenced, etc. The following example uses the result of function *InfoScrap* (pointer to a record) by dereferencing it:

```
var
    myHandle: Handle;
...
    myHandle := InfoScrap^.scrapHandle;
...
```

In Lightspeed Pascal, you'll need to use a temporary variable:

```
var
    myHandle: Handle;
    scrapInfo: PScrapStuff;
...
    scrapInfo := InfoScrap;
    myHandle := scrapInfo^.scrapHandle;
...
```


- **Type Casting of L-values.** In Lisa Pascal, type casts may be applied to l-values (variable-references appearing on the left-hand-side of an assignment, or as an actual **var** parameter) as well as r-values (variable-references appearing in expressions). In Lightspeed Pascal, only expressions can be type-cast. For example:

```
var
    myHandle: Handle;
    hTE: TEHandle;
...
    myHandle := hTE^.hText;
    CharsHandle(myHandle)^[0] := Chr($0D);
    DrawChar (CharsHandle(myHandle)^[0]);
    DisposHandle (myHandle);
...
```

In Lightspeed Pascal, you may need to change such declarations. In particular, it is best that variables such as *myHandle* above be declared as specific types, like *CharsHandle*, and cast to generic types, like *Handle*, as needed:

```
var
    myHandle: CharsHandle;
    hTE: TEHandle;
...
    myHandle := CharsHandle (hTE^.hText);
    myHandle^[0] := Chr($0D);
    DrawChar (myHandle^[0]);
    DisposHandle (Handle (myHandle));
...
```

- **Short-circuit Boolean Expression Evaluation.** In Lisa Pascal, the binary operators **&** (ampersand) and **|** (vertical bar) were introduced which cause minimum evaluation of boolean expressions. This allows the following statement:

```
if (p <> nil) & (p^.name = '') then
```

In Lightspeed Pascal, this must be written as:

```
if (p <> nil) then
    if (p^.name = '') then
```

Appendix C

QuickDraw

This appendix contains a reproduction of *QuickDraw: A Programmer's Guide*. The various constants, types, variables, procedures, and functions described here are accessible through Lightspeed Pascal.

The basic shape-drawing procedures described in Sections C.8.8 through C.8.11 takes a value of type *Rect* as its first parameter. As a convenience, the *Rect* parameter in each of these procedures may alternatively be replaced by four *integer* values giving the *top*, *left*, *bottom*, and *right* coordinates of the rectangle in that order.

There are three additional procedures, not normally a part of QuickDraw and thus not included in the main part of this appendix, that have been included in Lightspeed Pascal. These are as follows:

```
procedure DrawLine( a, b, c, d : integer );  
    {Draws a line from point (a,b) to point (c,d)}  
  
procedure PaintCircle( x, y, r : integer );  
    {Paints a circle centered at point (x,y) with radius r}  
  
procedure InvertCircle( x, y, r : integer );  
    {Inverts a circle centered at point (x,y) with radius r}
```

In general, any of the QuickDraw operations described in Sections C.8.3 through C.8.11 and C.8.18 are "safe" to use, as they have no effect beyond the Lightspeed Pascal Drawing window. The operations described in Section C.8.12 through C.8.17 are also "safe", except that they may cause your program to run out of memory. The operations described in Section C.8.1, C.8.2, and C.9 should be used only by knowledgeable programmers, since they can interfere with the normal operation of the Lightspeed Pascal environment. Also, as always, pointers and handles can get you into trouble if not used properly.

Appendix C

QuickDraw

C.1	About QuickDraw	C-5
C.1.1	How To Use QuickDraw	C-6
C.1.2	QuickDraw Data Types	C-6
C.2	The Mathematical Foundation of QuickDraw	C-7
C.2.1	The Coordinate Plane	C-7
C.2.2	Points	C-8
C.2.3	Rectangles	C-8
C.2.4	Region	C-10
C.3	Graphic Entities	C-12
C.3.1	The Bit Image	C-12
C.3.2	The Bitmap	C-13
C.3.3	Patterns	C-15
C.3.4	Cursors	C-15
C.4	The Drawing Environment: GrafPort	C-17
C.4.1	Pen Characteristics	C-20
C.4.2	Text Characteristics	C-21
C.5	Coordinates in GrafPorts	C-24
C.6	General Discussion of Drawing	C-26
C.6.1	Transfer Modes	C-27
C.6.2	Drawing in Color	C-29
C.7	Pictures and Polygons	C-29
C.7.1	Pictures	C-30
C.7.2	Polygons	C-31
C.8	QuickDraw Routines	C-32
C.8.1	GrafPort Routines	C-33
C.8.2	Cursor-Handling Routines	C-37
C.8.3	Pen and Line-Drawing Routines	C-38

C.8.5	Drawing in Color	C-43
C.8.6	Calculations with Points	C-44
C.8.7	Calculations with Rectangles	C-47
C.8.8	Graphic Operations on Rectangles	C-49
C.8.9	Graphic Operations on Ovals	C-50
C.8.10	Graphic Operations on Rounded-Corner Rectangles	C-51
C.8.11	Graphic Operations on Arcs and Wedges	C-53
C.8.12	Calculations with Regions	C-54
C.8.13	Graphic Operations on Regions	C-58
C.8.14	Bit Transfer Operations	C-59
C.8.15	Pictures	C-61
C.8.16	Calculations with Polygons	C-62
C.8.17	Graphic Operations on Polygons	C-64
C.8.18	Miscellaneous Utilities	C-65
C.9	Customizing QuickDraw Operations	C-66

QuickDraw

C.1 About QuickDraw

QuickDraw allows you to organize the Macintosh screen into a number of individual areas. Within each area you can draw many things, as illustrated in Figure C-1.

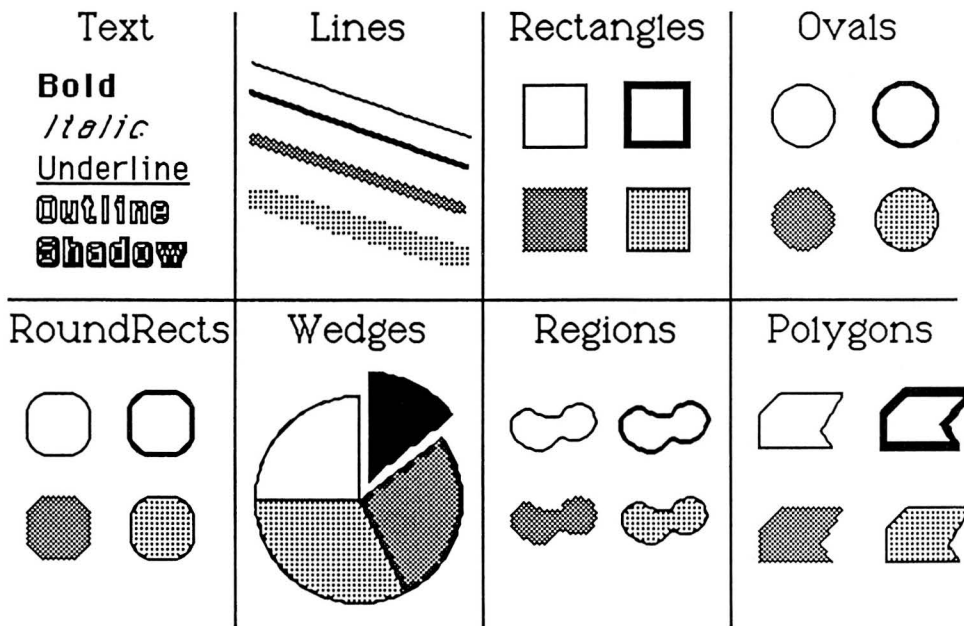


Figure C-1
Samples of QuickDraw's Abilities

You can draw:

- Text characters in a number of proportionally-spaced fonts, with variations that include boldfacing, italicizing, underlining, and outlining.
- Straight lines of any length and width.
- A variety of shapes, either solid or hollow, including: rectangles, with or without rounded corners; full circles and ovals or wedge-shaped sections; and polygons.

- Any other arbitrary shape or collection of shapes, again either solid or hollow.
- A picture consisting of any combination of the above items, with just a single procedure call.

In addition, QuickDraw has some other abilities that you won't find in many other graphics packages. These abilities take care of most of the "house-keeping"--the trivial but time-consuming and bothersome overhead that's necessary to keep things in order.

- The ability to define many distinct ports on the screen, each with its own complete drawing environment--its own coordinate system, drawing location, character set, location on the screen, and so on. You can easily switch from one such port to another.
- Full and complete clipping to arbitrary areas, so that drawing will occur only where you want. It's like a super-duper coloring book that won't let you color outside the lines. You don't have to worry about accidentally drawing over something else on the screen, or drawing off the screen and destroying memory.
- Off-screen drawing. Anything you can draw on the screen, you can draw into an off-screen buffer, so you can prepare an image for an output device without disturbing the screen, or you can prepare a picture and move it onto the screen very quickly.

And QuickDraw lives up to its name! It's very fast. The speed and responsiveness of the Macintosh user interface are due primarily to the speed of the QuickDraw package. You can do good-quality animation, fast interactive graphics, and complex yet speedy text displays using the full features of QuickDraw. This means you don't have to bypass the general-purpose QuickDraw routines by writing a lot of special routines to improve speed.

C.1.1 How To Use QuickDraw

QuickDraw includes only the graphics and utility procedures and functions you'll need to create graphics on the screen. Procedures for dealing with the mouse, cursors, and the keyboard are not described here: refer to *Inside Macintosh*. A summary of the Toolbox routines, types, etc., supported by Lightspeed Pascal is included in Appendix E.

C.1.2 QuickDraw Data Types

QuickDraw defines three general data types, *QDByte*, *QDPtr*, and *QDHandle*:

```

type
    QDByte = Signed Byte;   { -128..127 }
    QDPtr  = Ptr;           { blind pointer }
    QDHandle = Handle;      { blind handle }

```

Other types are described throughout this appendix in the sections in which they're relevant. For a summary of all QuickDraw types, see Appendix E.

C.2 The Mathematical Foundation of QuickDraw

To create graphics that are both precise and pretty requires not super-charged features but a firm mathematical foundation for the features you have. If the mathematics that underlie a graphics package are imprecise or fuzzy, the graphics will be, too. QuickDraw defines some clear mathematical constructs that are widely used in its procedures, functions, and data types: the *coordinate plane*, the *point*, the *rectangle*, and the *region*.

C.2.1 The Coordinate Plane

All information about location, placement, or movement that you give to QuickDraw is in terms of coordinates on a plane. The coordinate plane is a two-dimensional grid, as illustrated in Figure C-2.

There are two distinctive features of the QuickDraw coordinate plane:

- All grid coordinates are integers.
- All grid lines are infinitely thin.

These concepts are important! First, they mean that the QuickDraw plane is finite, not infinite (although it's very large). Horizontal coordinates range from -32768 to +32767, and vertical coordinates have the same range. Note that in Lighspeed Pascal, the lower bound of integers is -32767, and the coordinate plane is restricted accordingly.

Second, they mean that all elements represented on the coordinate plane are mathematically pure. Mathematical calculations using integer arithmetic will produce intuitively correct results. If you keep in mind that grid lines are infinitely thin, you'll never have "endpoint paranoia"--the confusion that results from not knowing whether that last dot is included in the line.

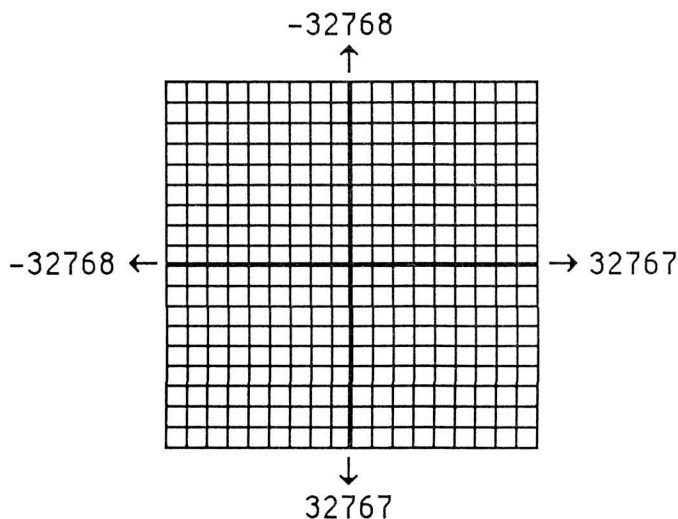


Figure C-2
The Coordinate Plane

C.2.2 Points

On the coordinate plane are 4,294,967,296 unique points. Each point is at the intersection of a horizontal grid line and a vertical grid line. As the grid lines are infinitely thin, a point is infinitely small. Of course there are more points on this grid than there are dots on the Macintosh screen: when using QuickDraw you associate small parts of the grid with areas on the screen, so that you aren't bound into an arbitrary, limited coordinate system.

The coordinate origin (0,0) is in the middle of the grid. Horizontal coordinates increase as you move from left to right, and vertical coordinates increase as you move from top to bottom. This is the way both a TV screen and a page of English text are scanned: from the top left to the bottom right.

You can store the coordinates of a point in a Pascal variable whose type is defined by QuickDraw. The type *Point* is a record of two integers, and has the following structure:

```

type
    VHSelect = (V,H);
    Point    = record
                case integer of
                    0: (v: integer;
                       h: integer);
                    1: (vh: array [VHSelect] of integer)
                end;

```

The variant part allows you to access the vertical and horizontal components of a point either individually or as an array. For example, if the variable *goodPt* were declared to be of type *Point*, the following would all refer to the coordinate parts of the point:

```

goodPt.v          goodPt.h
goodPt.vh[V]      goodPt.vh[H]

```

C.2.3 Rectangles

Any two points can define the top left and bottom right corners of a rectangle. As these points are infinitely small, the borders of the rectangle are infinitely thin (see Figure C-3).

Rectangles are used to define active areas on the screen, to assign coordinate systems to graphic entities, and to specify the locations and sizes for various drawing commands. QuickDraw also allows you to perform many mathematical calculations on rectangles--changing their sizes, shifting them around, and so on.

Note: Remember that rectangles, like points, are mathematical concepts that have no direct representation on the screen. The association between these

conceptual elements and their physical representations is made by a bitmap, described below.

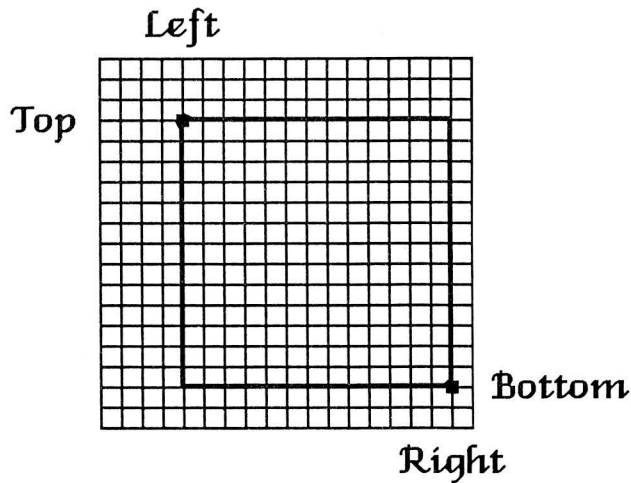


Figure C-3
A Rectangle

The data type for rectangles is *Rect*, and consists of four integers or two points:

```

type
  Rect = record
    case integer of
      0: (top: integer;
          left: integer;
          bottom: integer;
          right: integer);
      1: (topLeft: Point;
          botRight: Point)
    end;

```

Again, the record variant allows you to access a variable of type *Rect* either as four boundary coordinates or as two diagonally opposing corner points. Combined with the record variant for points, all of the following references to the rectangle named *bRect* are legal:

<i>bRect</i>		{type Rect}
<i>bRect.topLeft</i>	<i>bRect.botRight</i>	{type Point}
<i>bRect.top</i>	<i>bRect.left</i>	{type integer}
<i>bRect.topLeft.v</i>	<i>bRect.topLeft.h</i>	{type integer}
<i>bRect.topLeft.vh[V]</i>	<i>bRect.topLeft.vh[H]</i>	{type integer}

```

bRect.bottom          bRect.right          {type integer}
bRect.botRight.v      bRect.botRight.h      {type integer}
bRect.botRight.vh[V]  bRect.botRight.vh[H]  {type integer}

```

Warning: If the bottom coordinate of a rectangle is equal to or less than the top, or the right coordinate is equal to or less than the left, the rectangle is an empty rectangle (i.e., one that contains no bits).

C.2.4 Regions

Unlike most graphics packages that can manipulate only simple geometric structures (usually rectilinear, at that), QuickDraw can gather an arbitrary set of spatially coherent points into a structure called a region, and perform complex yet rapid manipulations and calculations on such structures. This remarkable feature not only will make your standard programs simpler and faster, but will let you perform operations that would otherwise be nearly impossible; it is fundamental to the Macintosh user interface.

You define a region by drawing lines, shapes such as rectangles and ovals, or even other regions. The outline of a region should be one or more closed loops. A region can be concave or convex, can consist of one area or many disjoint areas, and can even have "holes" in the middle. In Figure C-4, the region on the left has a hole in the middle, and the region on the right consists of two disjoint areas.

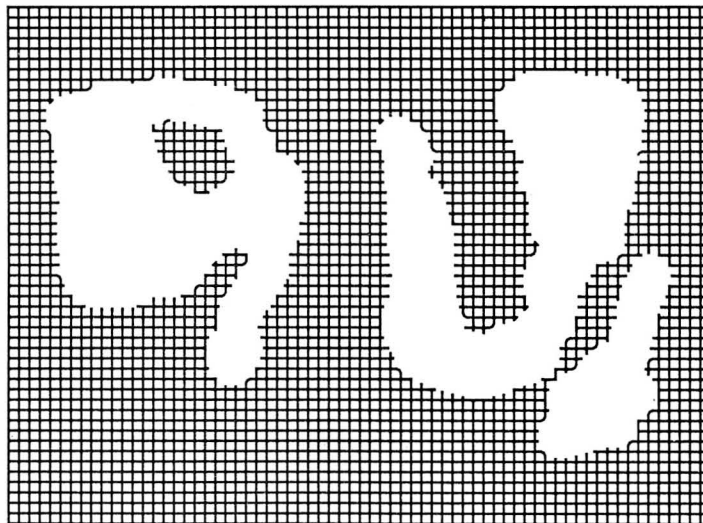


Figure C-4
Regions

Because a region can be any arbitrary area or set of areas on the coordinate plane, it takes a variable amount of information to store the outline of a region. The data structure for a region, therefore, is a variable-length entity with two fixed fields at the beginning, followed by a variable-length data field:

```

type
  Region = record
    rgnSize: integer;
    rgnBBox: Rect;
    {optional region definition data}
  end;

```

The *rgnSize* field contains the size, in bytes, of the region variable. The *rgnBBox* field is a rectangle which completely encloses the region.

The simplest region is a rectangle. In this case, the *rgnBBox* field defines the entire region, and there is no optional region data. For rectangular regions (or empty regions), the *rgnSize* field contains 10 (two bytes for *rgnSize*, plus eight for *rgnBBox*).

The region definition data for nonrectangular regions is stored in a compact way which allows for highly efficient access by QuickDraw procedures.

As regions are of variable size, they are stored dynamically on the heap, and the Operating System's memory management moves them around as their sizes change. Being dynamic, a region can be accessed only through a pointer; but when a region is moved, all pointers referring to it must be updated. For this reason, all regions are accessed through *handles*, which point to one master pointer which in turn points to the region.

```

type RgnPtr      = ^Region;
      RgnHandle = ^RgnPtr;

```

When the memory management relocates a region's data in memory, it updates only the *RgnPtr* master pointer to that region. The references through the master pointer can find the region's new home, but any references pointing directly to the region's previous position in memory would now point at dead bits. To access individual fields of a region, use the region handle and double indirection:

```

myRgn^.rgnSize      {size of region whose handle is myRgn}
myRgn^.rgnBBox      {rectangle enclosing the same region}
myRgn^.rgnBBox.top   {minimum vertical coordinate of all
                      points in the region}
myRgn^.rgnBBox      {syntactically incorrect; will not run if
                      myRgn is a rgnHandle}

```

Regions are created by a QuickDraw function which allocates space for the region, creates a master pointer, and returns a region handle. When you're done with a region, you dispose of it with another QuickDraw routine which frees up the space used by the region. Only these calls allocate or deallocate regions; do *not* use the Pascal procedure *new* to create a new region!

You specify the outline of a region with procedures that draw lines and shapes, as described in Section C.8, QuickDraw Routines. An example is given in the discussion of *CloseRgn* in Section C.8.12, Calculations with Regions.

Many calculations can be performed on regions. A region can be "expanded" or "shrunk" and, given any two regions, QuickDraw can find their union, intersection, difference, and exclusive-or; it can also determine whether a given point or rectangle intersects a given region, and so on. There is of course a set of graphic operations on regions to draw them on the screen.

C.3 Graphic Entities

Coordinate planes, points, rectangles, and regions are all good mathematical models, but they aren't really graphic elements--they don't have a direct physical appearance. Some graphic entities that do have a direct graphic interpretation are the *bit image*, *bitmap*, *pattern*, and *cursor*. This section describes the data structure of these graphic entities and how they relate to the mathematical constructs described above.

C.3.1 The Bit Image

A bit image is a collection of bits in memory which have a rectilinear representation. Take a collection of words in memory and lay them end to end so that bit 15 of the lowest-numbered word is on the left and bit 0 of the highest-numbered word is on the far right. Then take this array of bits and divide it, on word boundaries, into a number of equal-size rows. Stack these rows vertically so that the first row is on the top and the last row is on the bottom. The result is a matrix like the one shown in Figure C-5--rows and columns of bits, with each row containing the same number of bytes. The number of bytes in each row of the bit image is called the *row width* of that image.

A bit image can be stored in any static or dynamic variable, and can be of any length that is a multiple of the row width.

The Macintosh screen itself is one large visible bit image. There are 21,888 bytes of memory that are displayed as a matrix of 175,104 *pixels* on the screen, each bit corresponding to one pixel. If a bit's value is 0, its pixel is white; if the bit's value is 1, the pixel is black.

The screen is 342 pixels tall and 512 pixels wide, and the row width of its bit image is 64 bytes. Each pixel on the screen is square; there are 72 pixels per inch in each direction.

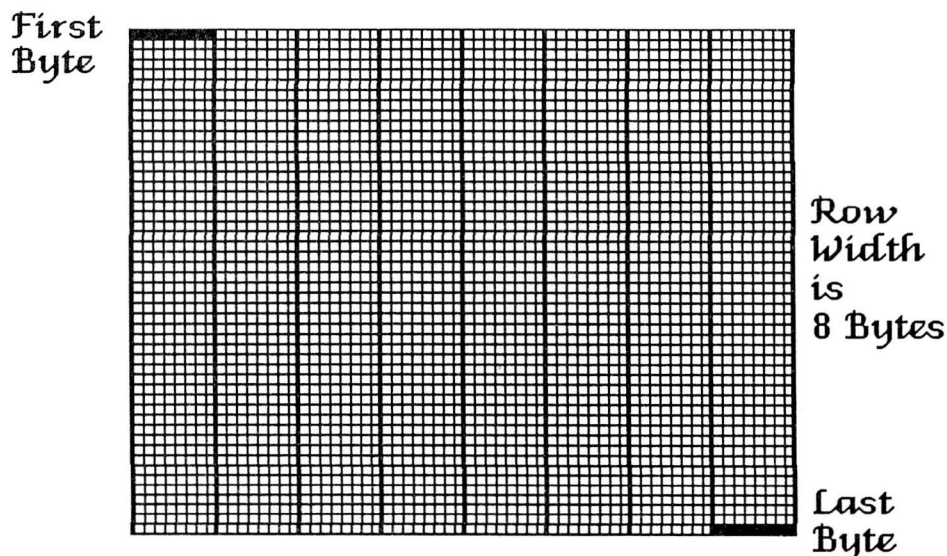


Figure C-5
A Bit Image

Note: Since each pixel on the screen represents one bit in a bit image, wherever this appendix says "bit", you can substitute "pixel" if the bit image is the Macintosh screen. Likewise, this appendix often refers to pixels on the screen where the discussion applies equally to bits in an off-screen bit image.

C.3.2 The Bitmap

When you combine the physical entity of a bit image with the conceptual entities of the coordinate plane and rectangle, you get a bitmap. A bitmap has three parts: a pointer to a bit image, the row width (in bytes) of that image, and a boundary rectangle which gives the bitmap both its dimensions and a coordinate system. Notice that a bitmap does not actually include the bits themselves: it points to them.

There can be several bitmaps pointing to the same bit image, each imposing a different coordinate system on it. This important feature is explained more fully in Section C.5, Coordinates in GrafPorts.

As shown in Figure C-6, the data structure of a bitmap is as follows:

```
type
  BitMap = record
    baseAddr: Ptr;
    rowBytes: integer;
    bounds: Rect
  end;
```

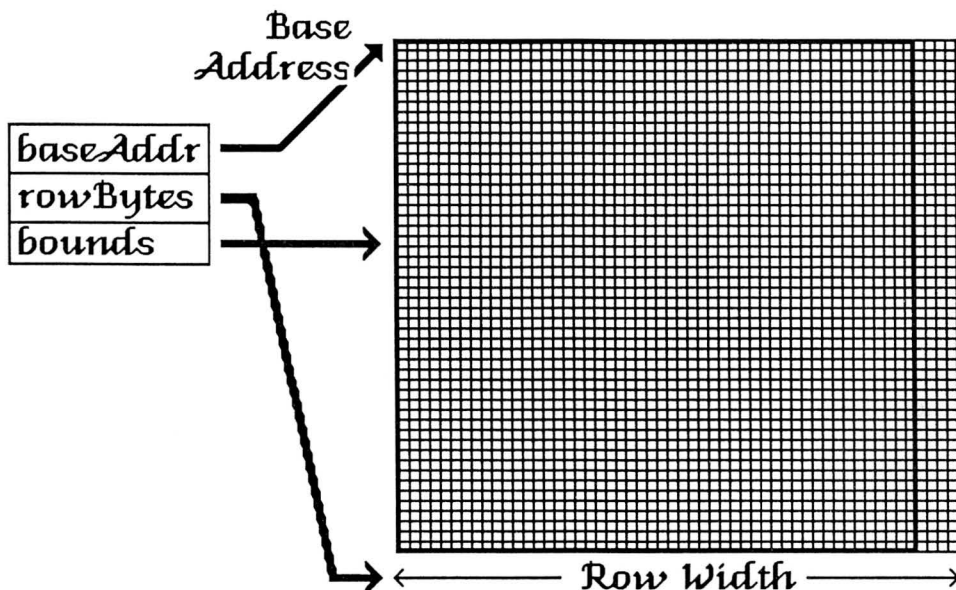


Figure C-6
A Bitmap

The *baseAddr* field is a pointer to the beginning of the bit image in memory, and the *rowBytes* field is the number of bytes in each row of the image. Both of these should always be even: a bitmap should always begin on a word boundary and contain an integral number of words in each row.

The *bounds* field is a boundary rectangle that both encloses the active area of the bit image and imposes a coordinate system on it. The relationship between the boundary rectangle and the bit image in a bitmap is simple yet very important. First, a few general rules:

- Bits in a bit image fall between points on the coordinate plane.
- A rectangle divides a bit image into two sets of bits: those bits inside the rectangle and those outside the rectangle.
- A rectangle that is *H* points wide and *V* points tall encloses exactly $(H-1) * (V-1)$ bits.

The top left corner of the boundary rectangle is aligned around the first bit in the bit image. The width of the rectangle determines how many bits of one row are logically owned by the bitmap; the relationship

$$8 * \text{map.rowBytes} \geq \text{map.bounds.right} - \text{map.bounds.left}$$

must always be true. The height of the rectangle determines how many rows of the image are logically owned by the bitmap. To ensure that the number of bits in the logical bitmap is not larger than the number of bits in the bit image, the bit image must be at least as big as

$$(\text{map.bounds.bottom} - \text{map.bounds.top}) * \text{map.rowBytes}$$

Normally, the boundary rectangle completely encloses the bit image: the width of the boundary rectangle is equal to the number of bits in one row of the image, and the height of the rectangle is equal to the number of rows in the image. If the rectangle is smaller than the dimensions of the image, the least significant bits in each row, as well as the last rows in the image, are not affected by any operations on the bitmap.

The bitmap also imposes a coordinate system on the image. Because bits fall between coordinate points, the coordinate system assigns integer values to the lines that border and separate bits, not to the bit positions themselves. For example, if a bitmap is assigned the boundary rectangle with corners (10,-8) and (34,8), the bottom right bit in the image will be between horizontal coordinates 33 and 34, and between vertical coordinates 7 and 8 (see Figure C-7).

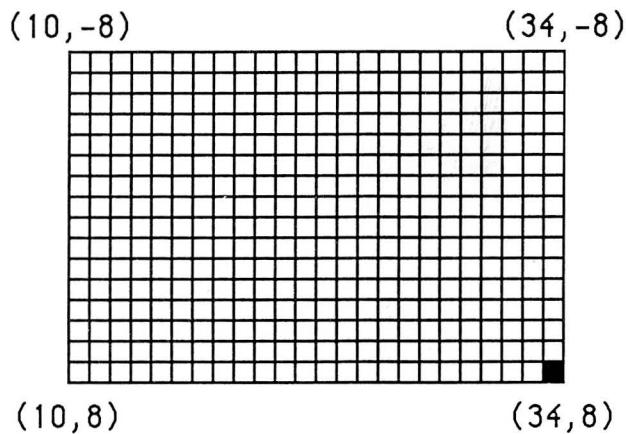


Figure C-7
Coordinates and Bitmaps

C.3.3 Patterns

A pattern is a 64-bit image, organized as an 8-by-8-bit rectangle, which is used to define a repeating design (such as stripes) or tone (such as gray). Patterns can be used to draw lines and shapes or to fill areas on the screen.

When a pattern is drawn, it is aligned such that adjacent areas of the same pattern in the same graphics port will blend with each other into a continuous, coordinated pattern. QuickDraw provides the predefined patterns *white*, *black*, *gray*, *ltGray*, and *dkGray*. Any other 64-bit variable or constant can be used as a pattern, too. The data type definition for a pattern is as follows:

```
type
    Pattern = packed array [0..7] of 0..255;
```

The row width of a pattern is 1 byte.

C.3.4 Cursors

A cursor is a small image that appears on the screen and is controlled by the mouse. (It appears only on the screen, and never in an off-screen bit image.)

Note: Other Macintosh documentation calls this image a "pointer", since it points to a location on the screen. To avoid confusion with the other meanings of "pointer" in this manual, the term "cursor" is used.

A cursor is defined as a 256-bit image, a 16-by-16-bit square. The row width of a cursor is 2 bytes. Figure C-8 illustrates four cursors.

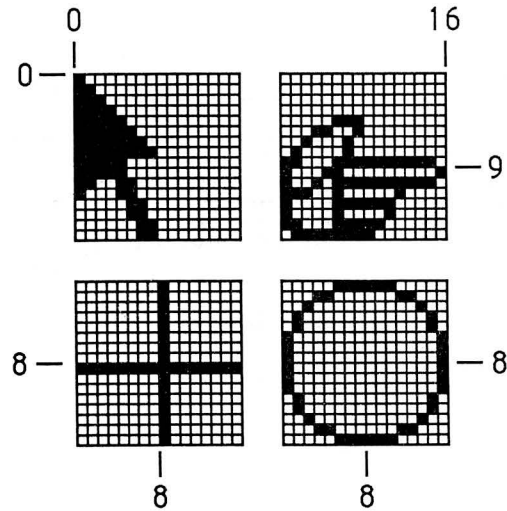


Figure C-8
Cursors

A cursor has three fields: a 16-word data field that contains the image itself, a 16-word mask field that contains information about the screen appearance of each bit of the cursor, and a *hotspot* point that aligns the cursor with the position of the mouse.

```

type
  Cursor = record
    data: array [0..15] of integer;
    mask: array [0..15] of integer;
    hotspot: Point
  end;

```

The data for the cursor must begin on a word boundary.

The cursor appears on the screen as a 16-by-16-bit square. The appearance of each bit of the rectangle is determined by the corresponding bits in the data and mask and, if the mask bit is 0, by the pixel "under" the cursor (the one already on the screen in the same position as this bit of the cursor):

Data	Mask	Resulting_pixel_on_screen
0	1	White
1	1	Black
0	0	Same as pixel under cursor
1	0	Inverse of pixel under cursor

Notice that if all mask bits are 0, the cursor is completely transparent, in that the image under the cursor can still be viewed: pixels under the white part of the cursor appear unchanged, while under the black part of the cursor, black pixels show through as white.

The hotspot aligns a point in the image (not a bit, a point!) with the mouse position. Imagine the rectangle with corners (0,0) and (16,16) framing the image, as in each of the examples in Figure

C-8; the hotspot is defined in this coordinate system. A hotspot of (0,0) is at the top left of the image. For the arrow in Figure C-8 to point to the mouse position, (0,0) would be its hotspot. A hotspot of (8,8) is in the exact center of the image; the center of the plus sign or oval in Figure C-8 would coincide with the mouse position if (8,8) were the hotspot for that cursor. Similarly, the hotspot for the pointing hand would be (16,9).

Whenever you move the mouse, the low-level interrupt-driven mouse routines move the cursor's hotspot to be aligned with the new mouse position.

QuickDraw supplies a predefined *arrow* cursor, an arrow pointing north- northwest.

C.4 The Drawing Environment: GrafPort

A *grafPort* is a complete drawing environment that defines how and where graphic operations will have their effect. It contains all the information about one instance of graphic output that is kept separate from all other instances. You can have many grafPorts open at once, and each one will have its own coordinate system, drawing pattern, background pattern, pen size and location, character font and style, and bitmap in which drawing takes place. You can instantly switch from one port to another. GrafPorts are the structures on which a program builds windows, which are fundamental to the Macintosh's "overlapping windows" user interface.

A grafPort is a dynamic data structure, defined as follows:

```
type GrafPtr    = ^GrafPort;
      GrafPort = record
                    device:      integer;
                    portBits:    BitMap;
                    portRect:    Rect;
                    visRgn:      RgnHandle;
                    clipRgn:     RgnHandle;
                    bkPat:       Pattern;
                    fillPat:     Pattern;
                    pnLoc:       Point;
                    pnSize:      Point;
                    pnMode:      integer;
                    pnPat:       Pattern;
                    pnVis:       integer;
                    txFont:      integer;
                    txFace:      Style;
                    txMode:      integer;
                    txSize:      integer;
                    spExtra:     longint;
                    fgColor:     longint;
                    bkColor:     longint;
                    colrBit:     integer;
                    patStretch:  integer;
                    picSave:     Handle;
                    rgnSave:     Handle;
                    polySave:     Handle;
                    grafProcs:   ProcsPtr
                end;
```

All QuickDraw operations refer to grafPorts via *grafPtrs*. You create a grafPort with the Pascal procedure *new* and use the resulting pointer in calls to QuickDraw. You could, of course, declare a static variable of type *GrafPort*, and obtain a pointer to that static structure (with the *@* operator), but as most grafPorts will be used dynamically, their data structures should be dynamic also.

Note: You can access all fields and subfields of a grafPort normally, but you should not store new values directly into them. QuickDraw has procedures for altering all fields of a grafPort, and using these procedures ensures that changing a grafPort produces no unusual side effects.

The *device* field of a grafPort indicates the output device that the grafPort will be using. This information is necessary because there are physical differences in the same font for different output devices. The default device number is 0, for the Macintosh screen.

The *portBits* field is the bitmap that points to the bit image to be used by the grafPort. All drawing that is done in this grafPort will take place in this bit image. The default bitmap uses the entire Macintosh screen as its bit image, with *rowBytes* of 64 and a boundary rectangle of (0,0,512,342). The bitmap may be changed to indicate a different structure in memory: all graphics procedures work in exactly the same way regardless of whether their effects are visible on the screen. A program can, for example, prepare an image to be printed on a printer without ever displaying the image on the screen, or develop a picture in an off-screen bitmap before transferring it to the screen. By altering the coordinates of the *portBits.bounds* rectangle, you can change the coordinate system of the grafPort; with a QuickDraw procedure call, you can set an arbitrary coordinate system for each grafPort, even if the different grafPorts all use the same bit image (e.g., the full screen).

The *portRect* field is a rectangle that defines a subset of the bitmap for use by the grafPort. Its coordinates are in the system defined by the *portBits.bounds* rectangle. All drawing done by the application occurs inside this rectangle. The *portRect* usually defines the "writable" interior area of a window, document, or other object on the screen. The default *portRect* is the entire screen.

The *visRgn* field indicates the region that is actually visible on the screen. It is reserved for use by Macintosh system software, and should be treated as read-only. The default *visRgn* is set to the *portRect*.

The *clipRgn* is an arbitrary region that the application can use to limit drawing to any region within the *portRect*. If, for example, you want to draw a half circle on the screen, you can set the *clipRgn* to half the square that would enclose the whole circle, and go ahead and draw the whole circle. Only the half within the *clipRgn* will actually be drawn in the grafPort. The default *clipRgn* is set arbitrarily large, and you have full control over its setting. Notice that unlike the *visRgn*, the *clipRgn* affects the image even if it is not displayed on the screen.

Figure C-9 illustrates a typical bitmap (as defined by *portBits*), *portRect*, *visRgn*, and *clipRgn*.

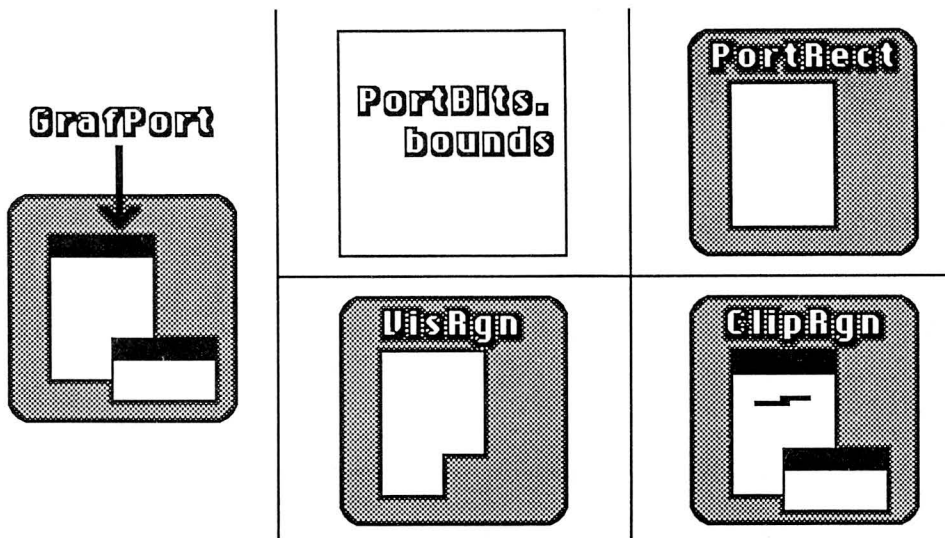


Figure C-9
GrafPort Regions

The *bkPat* and *fillPat* fields of a *grafPort* contain patterns used by certain QuickDraw routines. *BkPat* is the "background" pattern that is used when an area is erased or when bits are scrolled out of it. When asked to fill an area with a specified pattern, QuickDraw stores the given pattern in the *fillPat* field and then calls a low-level drawing routine which gets the pattern from that field. The various graphic operations are discussed in detail later in the descriptions of individual QuickDraw routines.

Of the next ten fields, the first five determine characteristics of the graphics pen, described in Section C.4.1, and the last five determine characteristics of any text that may be drawn, described in Section C.4.2.

The *fgColor*, *bkColor*, and *colrBit* fields contain values related to drawing in color, a capability that will be available in the future when Apple supports color output devices for the Macintosh. *FgColor* is the *grafPort*'s foreground color and *bkColor* is its background color. *ColrBit* tells the color imaging software which plane of the color picture to draw into. For more information, see Section C.6.2, Drawing in Color.

The *patStretch* field is used during output to a printer to expand patterns if necessary. The application should not change its value.

The *picSave*, *rgnSave*, and *polySave* fields reflect the state of picture, region, and polygon definition, respectively. To define a region, for example, you "open" it, call routines that draw it, and then "close" it. If no region is open, *rgnSave* contains *nil*; otherwise, it contains a handle to information related to the region definition. The application should not be concerned about exactly what information the handle leads to; you may, however, save the current value of *rgnSave*, set the field to *nil* to disable the region definition, and later restore it to the saved value to resume the region definition. The *picSave* and *polySave* fields work similarly for pictures and polygons.

Finally, the *grafProcs* field may point to a special data structure that the application stores into if it wants to customize QuickDraw drawing procedures or use QuickDraw in other advanced, highly specialized ways. (For more information, see Section C.9, Customizing QuickDraw Operations.) If *grafProcs* is *nil*, QuickDraw responds in the standard ways described in this appendix.

C.4.1 Pen Characteristics

The *pnLoc*, *pnSize*, *pnMode*, *pnPat*, and *pnVis* fields of a *grafPort* deal with the graphics pen. Each *grafPort* has one and only one graphics pen, which is used for drawing lines, shapes, and text. As illustrated in Figure C-10, the pen has four characteristics: a *location*, a *size*, a *drawing mode*, and a *drawing pattern*.

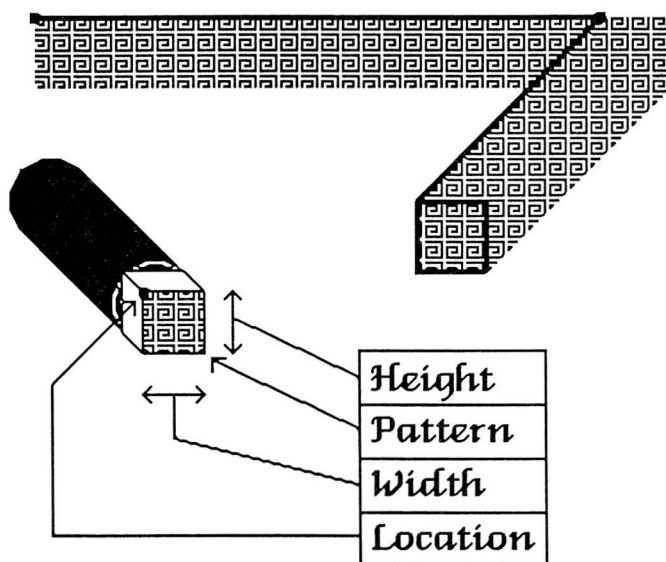


Figure C-10
A Graphics Pen

The pen location (*pnLoc*) is a point in the coordinate system of the *grafPort*, and is where QuickDraw will begin drawing the next line, shape, or character. It can be anywhere on the coordinate plane: there are no restrictions on the movement or placement of the pen. Remember that the pen location is a point on the coordinate plane, not a pixel in a bit image!

The pen is rectangular in shape, and has a user-definable width and height (*pnSize*). The default size is a 1-by-1-bit square; the width and height can range from (0,0) to (32767,32767). If either the pen width or the pen height is less than 1, the pen will not draw on the screen.

- The pen appears as a rectangle with its top left corner at the pen location; it hangs below and to the right of the pen location.

The *pnMode* and *pnPat* fields of a *grafPort* determine how the bits under the pen are affected when lines or shapes are drawn. The *pnPat* is a pattern that is used as the "ink" in the pen. This pattern, like all other patterns drawn in the *grafPort*, is always aligned with the port's coordinate

system: the top left corner of the pattern is aligned with the top left corner of the *portRect*, so that adjacent areas of the same pattern will blend into a continuous, coordinated pattern. Five patterns are predefined (white, black, and three shades of gray); you can also create your own pattern and use it as the *pnPat*. (A utility procedure, called *StuffHex*, allows you to fill patterns easily.)

The *pnMode* field determines how the pen pattern is to affect what's already on the bitmap when lines or shapes are drawn. When the pen draws, QuickDraw first determines what bits of the bitmap will be affected and finds their corresponding bits in the pattern. It then does a bit-by-bit evaluation based on the pen mode, which specifies one of eight boolean operations to perform. The resulting bit is placed into its proper place in the bitmap. The pen modes are described in Section C.6.1, Transfer Modes.

The *pnVis* field determines the pen's visibility, that is, whether it draws on the screen. For more information, see the descriptions of *HidePen* and *ShowPen* in Section C.8.3, Pen and Line-Drawing Routines.

C.4.2 Text Characteristics

The *txFont*, *txFace*, *txMode*, *txSize*, and *spExtra* fields of a *grafPort* determine how text will be drawn--the font, style, and size of characters and how they will be placed on the bitmap.

QuickDraw can draw characters as quickly and easily as it draws lines and shapes, and in many prepared fonts. Figure C-11 shows two QuickDraw characters and some terms you should become familiar with.

QuickDraw can display characters in any size, as well as boldfaced, italicized, outlined, or shadowed, all without changing fonts. It can also underline the characters, or draw them closer together or farther apart.

The *txFont* field is a font number that identifies the character font to be used in the *grafPort*. The font number 0 represents the system font, and is the default established by *OpenPort*.

A character font is defined as a collection of bit images: these images make up the individual characters of the font. The characters can be of unequal widths, and they're not restricted to their "cells": the lower curl of a lowercase j, for example, can stretch back under the previous character (typographers call this *Kerning*). A font can consist of up to 256 distinct characters, yet not all characters need be defined in a single font. Each font contains a *missing symbol* to be drawn in case of a request to draw a character that is missing from the font.

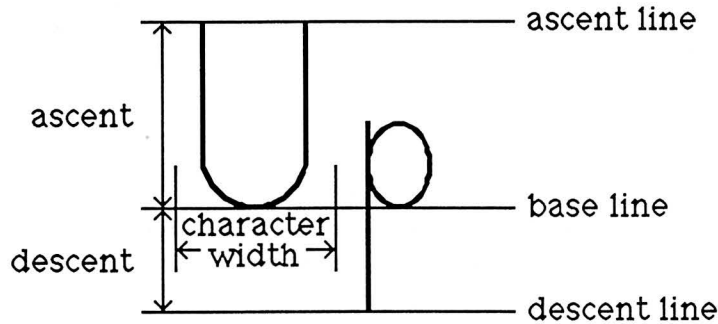


Figure C-11
QuickDraw Characters

The `txFace` field controls the appearance of the font with values from the set defined by the `Style` data type:

```
type
    StyleItem = (bold,    italic,    underline, outline,
                shadow, condense, extend);

    Style = set of StyleItem;
```

You can apply these either alone or in combination (see Figure C-12). Most combinations usually look good only for large fonts.

If you specify *bold*, each character is repeatedly drawn one bit to the right an appropriate number of times for extra thickness.

Italic adds an italic slant to the characters. Character bits above the base line are skewed right; bits below the base line are skewed left.

Underline draws a line below the base line of the characters. If part of a character descends below the base line, the underline is not drawn through the pixel on either side of the descending part.

Normal Characters
Bold Characters
Italic Characters
Underlined Characters
Outlined Characters
Shadowed Characters
 Condensed Characters
 Extended Characters
Bold Italic Characters
Bold Outlined Underlined
 ...and in other fonts, too!

Figure C-12
 Character Styles

You may specify either *outline* or *shadow*. *Outline* makes a hollow, outlined character rather than a solid one. With *shadow*, not only is the character hollow and outlined, but the outline is thickened below and to the right of the character to achieve the effect of a shadow. If you specify *bold* along with *outline* or *shadow*, the hollow part of the character is widened.

Condense and *extend* affect the horizontal distance between all characters, including spaces. *Condense* decreases the distance between characters and *extend* increases it, by an amount which QuickDraw determines is appropriate.

The *txMode* field controls the way characters are placed on a bit image. It functions much like a *pnMode*: when a character is drawn, QuickDraw determines which bits of the bit image will be affected, does a bit-by-bit comparison based on the mode, and stores the resulting bits into the bit image. These modes are described in Section C.6.1, Transfer Modes. Only three of them--*srcOr*, *srcXor*, and *srcBic*--should be used for drawing text.

The *txSize* field specifies the type size for the font, in points (where "point" here is a typographical term meaning approximately 1/72 inch). Any size may be specified. If QuickDraw does not have the font in a specified size, it will scale a size it does have as necessary to produce the size desired. A value of 0 in this field directs QuickDraw to choose the system font size, 12 point, or to scale to that size if necessary.

Finally, the *spExtra* field is useful when a line of characters is to be drawn justified such that it is aligned with both a left and a right margin (sometimes called "full justification"). *SpExtra* is a fixed-point number (see Section 10.6.4.5) that represents the average number of pixels by which each space character should be widened to fill out the line.

C.5 Coordinates in GrafPorts

Each grafPort has its own *local* coordinate system. All fields in the grafPort are expressed in these coordinates, and all calculations and actions performed in QuickDraw use the local coordinate system of the currently selected port.

Two things are important to remember:

- Each grafPort maps a portion of the coordinate plane into a similarly- sized portion of a bit image.
- The *portBits.bounds* rectangle defines the local coordinates for a grafPort.

The top left corner of *portBits.bounds* is always aligned around the first bit in the bit image; the coordinates of that corner "anchor" a point on the grid to that bit in the bit image. This forms a common reference point for multiple grafPorts using the same bit image (such as the screen). Given the rectangle *portBits.bounds* for each port, you know that their top left corners coincide.

The interrelationship between the *portBits.bounds* and *portRect* rectangles is very important. As the *portBits.bounds* rectangle establishes a coordinate system for the port, the *portRect* rectangle indicates the section of the coordinate plane (and thus the bit image) that will be used for drawing. The *portRect* usually falls inside the *portBits.bounds* rectangle, but it's not required to do so.

When a new grafPort is created, its bitmap is set to point to the entire Macintosh screen, and both the *portBits.bounds* and the *portRect* rectangles are set to 512-by-342-bit rectangles, with the point (0,0) at the top left corner of the screen.

You can redefine the local coordinates of the top left corner of the grafPort's *portRect*, using the *SetOrigin* procedure. This changes the local coordinate system of the grafPort, recalculating the coordinates of all points in the grafPort to be relative to the new corner coordinates. For example, consider these procedure calls:

```
SetPort(gamePort);  
SetOrigin(40,80);
```

The call to *SetPort* sets the current grafPort to *gamePort*; the call to *SetOrigin* changes the local coordinates of the top left corner of that port's *portRect* to (40,80) (see Figure C-13).

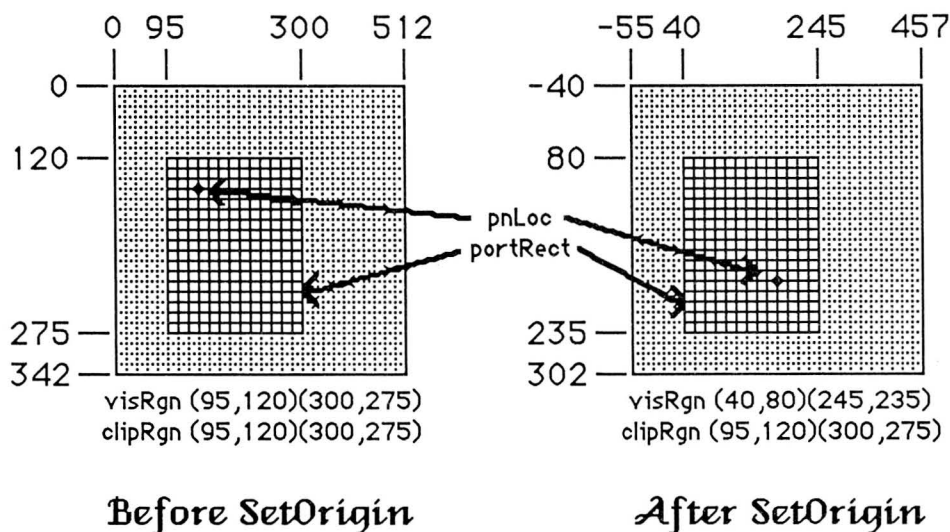


Figure C-13
Changing Local Coordinates

This recalculates the coordinate components of the following elements:

```
gamePort^.portBits.bounds  gamePort^.portRect
gamePort^.visRgn
```

These elements are always kept "in sync", so that all calculations, comparisons, or operations that seem right, work right.

Notice that when the local coordinates of a grafPort are offset, the *visRgn* of that port is offset also, but the *clipRgn* is not. A good way to think of it is that if a document is being shown inside a grafPort, the document "sticks" to the coordinate system, and the port's structure "sticks" to the screen. Suppose, for example, that the *visRgn* and *clipRgn* in Figure C-13 before *SetOrigin* are the same as the *portRect*, and a document is being shown. After the *SetOrigin* call, the top left corner of the *clipRgn* is still (95,120), but this location has moved down and to the right, and the location of the pen within the document has similarly moved. The locations of *portBits.bounds*, *portRect*, and *visRgn* did not change; their coordinates were offset. As always, the top left corner of *portBits.bounds* remains aligned around the first bit in the bit image (the first pixel on the screen).

If you are moving, comparing, or otherwise dealing with mathematical items in different grafPorts (for example, finding the intersection of two regions in two different grafPorts), you must adjust to a common coordinate system before you perform the operation. A QuickDraw procedure, *LocalToGlobal*, lets you convert a point's local coordinates to a *global* system where the top left corner of the bit image is (0,0); by converting the various local coordinates to global coordinates, you can compare and mix them with confidence. For more information, see the description of this procedure in Section C.8.6, Calculations with Points.

C.6 General Discussion of Drawing

Drawing occurs:

- Always inside a `grafPort`, in the bit image and coordinate system defined by the `grafPort`'s bitmap.
- Always within the intersection of the `grafPort`'s `portBits.bounds` and `portRect`, and clipped to its `visRgn` and `clipRgn`.
- Always at the `grafPort`'s pen location.
- Usually with the `grafPort`'s pen size, pattern, and mode.

With QuickDraw procedures, you can draw lines, shapes, and text. Shapes include rectangles, ovals, rounded-corner rectangles, wedge-shaped sections of ovals, regions, and polygons.

Lines are defined by two points: the current pen location and a destination location. When drawing a line, QuickDraw moves the top left corner of the pen along the mathematical trajectory from the current location to the destination. The pen hangs below and to the right of the trajectory (see Figure C-14).

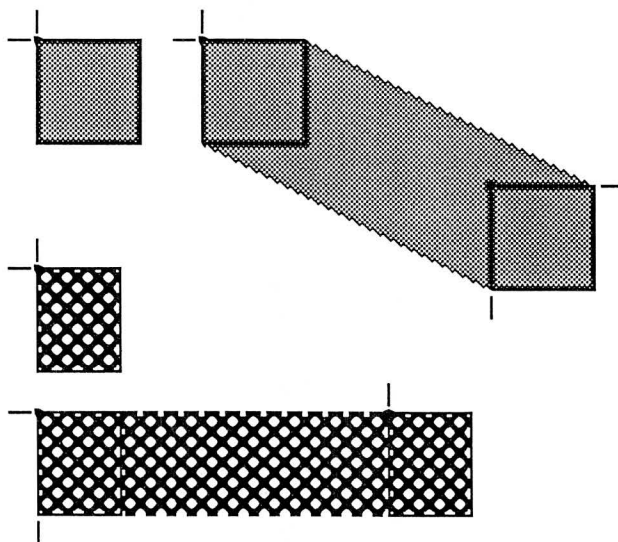


Figure C-14
Drawing Lines

Note: No mathematical element (such as the pen location) is ever affected by clipping; clipping only determines what appears where in the bit image. If you draw a line to a location outside your `grafPort`, the pen location will move there, but only the portion of the line that is inside the port will actually be drawn. This is true for all drawing procedures.

Rectangles, ovals, and rounded-corner rectangles are defined by two corner points. The shapes always appear inside the mathematical rectangle defined by the two points. A region is defined in

a more complex manner, but also appears only within the rectangle enclosing it. Remember, these enclosing rectangles have infinitely thin borders and are not visible on the screen.

As illustrated in Figure C-15, shapes may be drawn either *solid* (filled in with a pattern) or *framed* (outlined and hollow).

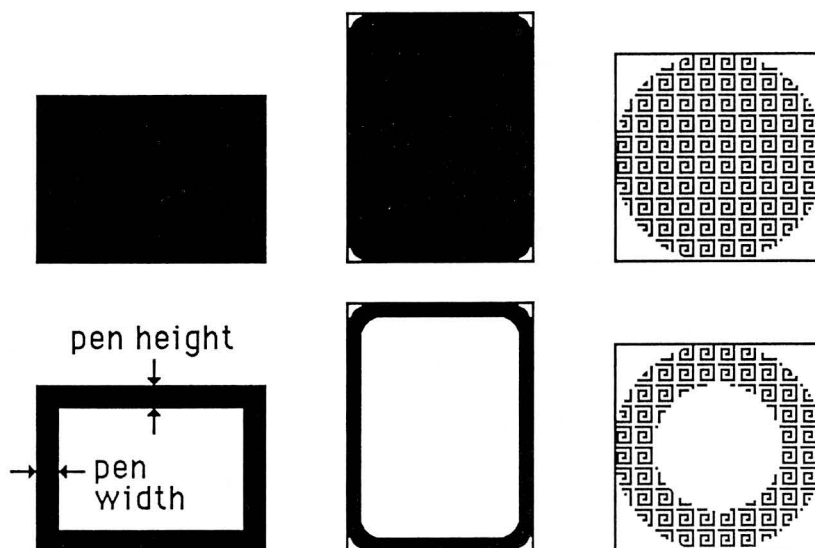


Figure C-15
Solid Shapes and Framed Shapes

In the case of framed shapes, the outline appears completely within the enclosing rectangle--with one exception--and the vertical and horizontal thickness of the outline is determined by the pen size. The exception is polygons, as discussed in Section C.7.2, Polygons.

The pen pattern is used to fill in the bits that are affected by the drawing operation. The pen mode defines how those bits are to be affected by directing QuickDraw to apply one of eight boolean operations to the bits in the shape and the corresponding pixels on the screen.

Text drawing does not use the *pnSize*, *pnPat*, or *pnMode*, but it does use the *pnLoc*. Each character is placed to the right of the current pen location, with the left end of its base line at the pen's location. The pen is moved to the right to the location where it will draw the next character. No wrap or carriage return is performed automatically.

The method QuickDraw uses in placing text is controlled by a mode similar to the pen mode. This is explained in Section C.6.1, Transfer Modes. Clipping of text is performed in exactly the same manner as all other clipping in QuickDraw.

C.6.1 Transfer Modes

When lines or shapes are drawn, the *pnMode* field of the *grafPort* determines how the drawing is to appear in the port's bit image; similarly, the *txMode* field determines how text is to appear. There is also a QuickDraw procedure that transfers a bit image from one bitmap to another, and

this procedure has a mode parameter that determines the appearance of the result. In all these cases, the mode, called a *transfer mode*, specifies one of eight boolean operations: for each bit in the item to be drawn, QuickDraw finds the corresponding bit in the destination bit image, performs the boolean operation on the pair of bits, and stores the resulting bit into the bit image.

There are two types of transfer mode:

- *Pattern transfer modes*, for drawing lines or shapes with a pattern.
- *Source transfer modes*, for drawing text or transferring any bit image between two bitmaps.

For each type of mode, there are four basic operations--*Copy*, *Or*, *Xor*, and *Bic*. The *Copy* operation simply replaces the pixels in the destination with the pixels in the pattern or source, "painting" over the destination without regard for what is already there. The *Or*, *Xor*, and *Bic* operations leave the destination pixels under the white part of the pattern or source unchanged, and differ in how they affect the pixels under the black part: *Or* replaces those pixels with black pixels, thus "overlaying" the destination with the black part of the pattern or source; *Xor* inverts the pixels under the black part; and *Bic* erases them to white.

Each of the basic operations has a variant in which every pixel in the pattern or source is inverted before the operation is performed, giving eight operations in all. Each mode is defined by name as a constant in QuickDraw (see Figure C-16).

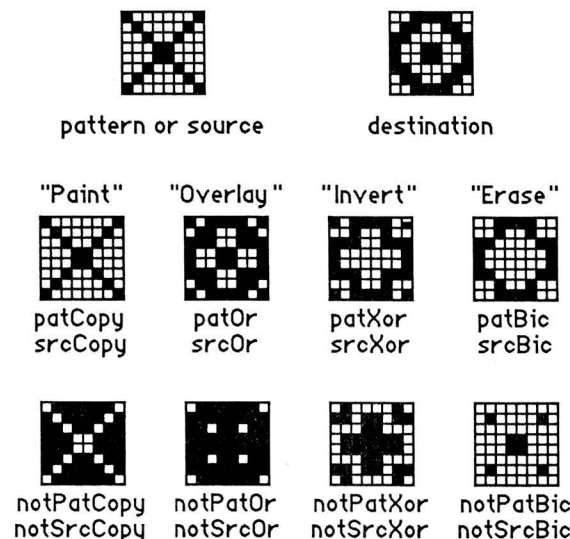


Figure C-16
Transfer Modes

Pattern	Source	Action on each pixel in destination:	
transfer mode	transfer mode	If black pixel in pattern_or_source	If white pixel in pattern_or_source
<i>patCopy</i>	<i>srcCopy</i>	Force black	Force white
<i>patOr</i>	<i>srcOr</i>	Force black	Leave alone
<i>patXor</i>	<i>srcXor</i>	Invert	Leave alone
<i>patBic</i>	<i>srcBic</i>	Force white	Leave alone
<i>notPatCopy</i>	<i>notSrcCopy</i>	Force white	Force black
<i>notPatOr</i>	<i>notSrcOr</i>	Leave alone	Force black
<i>notPatXor</i>	<i>notSrcXor</i>	Leave alone	Invert
<i>notPatBic</i>	<i>notSrcBic</i>	Leave alone	Force white

C.6.2 Drawing in Color

Currently you can only look at QuickDraw output on a black-and-white screen or printer. Eventually, however, Apple will support color output devices. If you want to set up your application now to produce color output in the future, you can do so by using QuickDraw procedures to set the foreground color and the background color. Eight standard colors may be specified with the following predefined constants: *blackColor*, *whiteColor*, *redColor*, *greenColor*, *blueColor*, *cyanColor*, *magentaColor*, and *yellowColor*. Initially, the foreground color is *blackColor* and the background color is *whiteColor*. If you specify a color other than *whiteColor*, it will appear as black on a black-and-white output device.

To apply the table above (in Section C.6.1) to drawing in color, make the following translation: where the table shows "Force black", read "Force foreground color", and where it shows "Force white", read "Force background color". When you eventually receive the color output device, you'll find out the effect of inverting a color on it.

Note: QuickDraw can support output devices that have up to 32 bits of color information per pixel. A color picture may be thought of, then, as having up to 32 planes. At any one time, QuickDraw draws into only one of these planes. A QuickDraw routine called by the color-imaging software specifies which plane.

C.7 Pictures and Polygons

QuickDraw lets you save a sequence of drawing commands and "play them back" later with a single procedure call. There are two such mechanisms: one for drawing any picture to scale in a destination rectangle that you specify, and another for drawing polygons in all the ways you can draw other shapes in QuickDraw.

C.7.1 Pictures

A *picture* in QuickDraw is a transcript of calls to routines which draw something--anything--on a bitmap. Pictures make it easy for one program to draw something defined in another program, with great flexibility and without knowing the details about what's being drawn.

For each picture you define, you specify a rectangle that surrounds the picture; this rectangle is called the *picture frame*. When you later call the procedure that draws the saved picture, you supply a destination rectangle, and QuickDraw scales the picture so that its frame is completely aligned with the destination rectangle. Thus, the picture may be expanded or shrunk to fit its destination rectangle. For example, if the picture is a circle inside a square picture frame, and the destination rectangle is not square, the picture is drawn as an oval.

Since a picture may include any sequence of drawing commands, its data structure is a variable-length entity. It consists of two fixed fields followed by a variable-length data field:

```
type
    Picture = record
        picSize:   integer;
        picFrame:  Rect;
        {picture definition data}
    end;
```

The *picSize* field contains the size, in bytes, of the picture variable. The *picFrame* field is the picture frame which surrounds the picture and gives a frame of reference for scaling when the picture is drawn. The rest of the structure contains a compact representation of the drawing commands that define the picture.

All pictures are accessed through handles, which point to one master pointer which in turn points to the picture.

```
type
    PicPtr = ^Picture;
    PicHandle = ^PicPtr;
```

To define a picture, you call a QuickDraw function that returns a picture handle and then call the routines that draw the picture. There is a procedure to call when you've finished defining the picture, and another for when you're done with the picture altogether.

QuickDraw also allows you to intersperse *picture comments* with the definition of a picture. These comments, which do not affect the picture's appearance, may be used to provide additional information about the picture when it's played back. This is especially valuable when pictures are transmitted from one application to another. There are two standard types of comment which, like parentheses, serve to group drawing commands together (such as all the commands that draw a particular part of a picture):

```
const
    picLParen = 0;
    picRParen = 1;
```


The application defining the picture can use these standard comments as well as comments of its own design.

To include a comment in the definition of a picture, the application calls a QuickDraw procedure that specifies the comment with three parameters: the comment kind, which identifies the type of comment; a handle to additional data if desired; and the size of the additional data, if any. When playing back a picture, QuickDraw passes any comments in the picture's definition to a low-level procedure accessed indirectly through the *grafProcs* field of the *grafPort* (see Section C.9, Customizing QuickDraw Operations, for more information). To process comments, the application must include a procedure to do the processing and store a pointer to it in the data structure pointed to by the *grafProcs* field.

Note: The standard low-level procedure for processing picture comments simply ignores all comments.

C.7.2 Polygons

Polygons are similar to pictures in that you define them by a sequence of calls to QuickDraw routines. They are also similar to other shapes that QuickDraw knows about, since there is a set of procedures for performing graphic operations and calculations on them.

A *polygon* is simply any sequence of connected lines (see Figure C-17). You define a polygon by moving to the starting point of the polygon and drawing lines from there to the next point, from that point to the next, and so on.

The data structure for a polygon is a variable-length entity. It consists of two fixed fields followed by a variable-length array:

```
type
  Polygon = record
    polySize:    integer;
    polyBBox:    Rect;
    polyPoints:  array [0..0] of Point
  end;
```

The *polySize* field contains the size, in bytes, of the polygon variable. The *polyBBox* field is a rectangle which just encloses the entire polygon. The *polyPoints* array expands as necessary to contain the points of the polygon-- the starting point followed by each successive point to which a line is drawn.

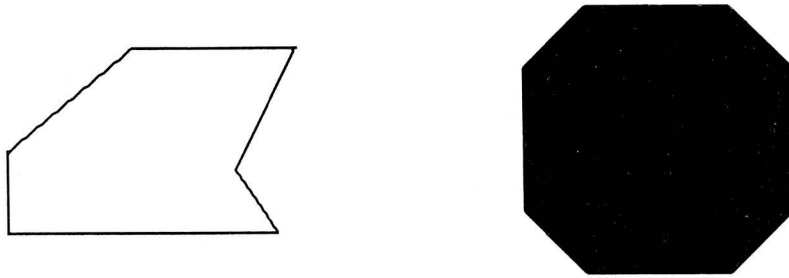


Figure C-17
Polygons

```
type  
  PolyPtr      = ^Polygon;  
  PolyHandle = ^PolyPtr;
```

To define a polygon, you call a QuickDraw function that returns a polygon handle and then form the polygon by calling procedures that draw lines. You call a procedure when you've finished defining the polygon, and another when you're done with the polygon altogether.

Just as for other shapes that QuickDraw knows about, there is a set of graphic operations on polygons to draw them on the screen. QuickDraw draws a polygon by moving to the starting point and then drawing lines to the remaining points in succession, just as when the routines were called to define the polygon. In this sense it "plays back" those routine calls. As a result, polygons are not treated exactly the same as other QuickDraw shapes. For example, the procedure that frames a polygon draws outside the actual boundary of the polygon, because QuickDraw line-drawing routines draw below and to the right of the pen location. The procedures that fill a polygon with a pattern, however, stay within the boundary of the polygon; they also add an additional line between the ending point and the starting point if those points are not the same, to complete the shape.

There is also a difference in the way QuickDraw scales a polygon and a similarly-shaped region if it's being drawn as part of a picture: when stretched, a slanted line is drawn more smoothly if it's part of a polygon rather than a region. You may find it helpful to keep in mind the conceptual difference between polygons and regions: a polygon is treated more as a continuous shape, a region more as a set of bits.

C.8 QuickDraw Routines

This section describes all the procedures and functions in QuickDraw, their parameters, and their operation. Note that the actual procedure and function declarations are given here, rather than the BNF notation or syntax diagrams used elsewhere in this manual.

C.8.1 GrafPort Routines

procedure *InitGraf*(*globalPtr* : *Ptr*);

InitGraf initializes QuickDraw. It is called automatically by Lightspeed Pascal's initialization routine; you need not call it again. It initializes the QuickDraw global variables listed below.

Variable	Type	Initial_setting
<i>thePort</i>	<i>GrafPtr</i>	<i>nil</i>
<i>white</i>	<i>Pattern</i>	<i>all-white pattern</i>
<i>black</i>	<i>Pattern</i>	<i>all-black pattern</i>
<i>gray</i>	<i>Pattern</i>	<i>50% gray pattern</i>
<i>ltGray</i>	<i>Pattern</i>	<i>25% gray pattern</i>
<i>dkGray</i>	<i>Pattern</i>	<i>75% gray pattern</i>
<i>arrow</i>	<i>Cursor</i>	<i>pointing arrow cursor</i>
<i>screenBits</i>	<i>BitMap</i>	<i>Macintosh screen, (0,0,512,342)</i>
<i>randSeed</i>	<i>longint</i>	<i>1</i>

The *globalPtr* parameter tells QuickDraw where to store its global variables, beginning with *thePort*. From Pascal programs, this parameter should always be set to *@thePort*.

Note: To initialize the cursor, call *InitCursor* (described in Section C.8.2, Cursor-Handling Routines).

procedure *OpenPort*(*gp* : *GrafPtr*);

OpenPort allocates space for the given grafPort's *visRgn* and *clipRgn*, initializes the fields of the grafPort as indicated below, and makes the grafPort the current port (see *SetPort*, below). You must call *OpenPort* before using any grafPort; first create a grafPtr with *new*, then use that grafPtr in the *OpenPort* call.

Field	Type	Initial_setting
<i>device</i>	<i>integer</i>	<i>0 (Macintosh screen)</i>
<i>portBits</i>	<i>BitMap</i>	<i>screenBits (see InitGraf)</i>
<i>portRect</i>	<i>Rect</i>	<i>screenBits.bounds (0,0,512,342)</i>
<i>visRgn</i>	<i>RgnHandle</i>	<i>handle to the rectangular region (0,0,512,342)</i>
<i>clipRgn</i>	<i>RgnHandle</i>	<i>handle to the rectangular region (-30000, -30000, 30000, 30000)</i>
<i>bkPat</i>	<i>Pattern</i>	<i>white</i>
<i>fillPat</i>	<i>Pattern</i>	<i>black</i>
<i>pnLoc</i>	<i>Point</i>	<i>(0,0)</i>
<i>pnSize</i>	<i>Point</i>	<i>(1,1)</i>
<i>pnMode</i>	<i>integer</i>	<i>patCopy</i>
<i>pnPat</i>	<i>Pattern</i>	<i>black</i>
<i>pnVis</i>	<i>integer</i>	<i>0 (visible)</i>
<i>txFont</i>	<i>integer</i>	<i>0 (system font)</i>
<i>txFace</i>	<i>Style</i>	<i>normal</i>
<i>txMode</i>	<i>integer</i>	<i>srcOr</i>

<i>txSize</i>	<i>integer</i>	0 (QuickDraw decides)
<i>spExtra</i>	<i>longint</i>	0
<i>fgColor</i>	<i>longint</i>	<i>blackColor</i>
<i>bkColor</i>	<i>longint</i>	<i>whiteColor</i>
<i>colrBit</i>	<i>integer</i>	0
<i>patStretch</i>	<i>integer</i>	0
<i>picSave</i>	<i>Handle</i>	nil
<i>rgnSave</i>	<i>Handle</i>	nil
<i>polySave</i>	<i>Handle</i>	nil
<i>grafProcs</i>	<i>ProcsPtr</i>	nil

procedure *InitPort*(*gp* : *GrafPtr*);

Given a pointer to a *grafPort* that has been opened with *OpenPort*, *InitPort* reinitializes the fields of the *grafPort* and makes it the current port (if it's not already).

Note: *InitPort* does everything *OpenPort* does except allocate space for the *visRgn* and *clipRgn*.

procedure *ClosePort*(*gp* : *GrafPtr*);

ClosePort deallocates the space occupied by the given *grafPort*'s *visRgn* and *clipRgn*. When you are completely through with a *grafPort*, call this procedure.

Warnings: If you do not call *ClosePort* before disposing of the *grafPort*, the memory used by the *visRgn* and *clipRgn* will be unrecoverable.

After calling *ClosePort*, be sure not to use any copies of the *visRgn* or *clipRgn* handles that you may have made.

procedure *SetPort*(*gp* : *GrafPtr*);

SetPort sets the *grafPort* indicated by *gp* to be the current port. The global pointer *thePort* always points to the current port. All QuickDraw drawing routines affect the bitmap *thePort*[^].*portBits* and use the local coordinate system of *thePort*[^]. Note that *OpenPort* and *InitPort* do a *SetPort* to the given port.

Warning: Never do a *SetPort* to a port that has not been opened with *OpenPort*.

Each port possesses its own pen and text characteristics which remain unchanged when the port is not selected as the current port.

procedure *GetPort*(**var** *gp* : *GrafPtr*);

GetPort returns a pointer to the current *grafPort*. If you have a program that draws into more than one *grafPort*, it's extremely useful to have each procedure save the current *grafPort* (with *GetPort*), set its own *grafPort*, do drawing or calculations, and then restore the previous *grafPort* (with *SetPort*). The pointer to the current *grafPort* is also available through the global pointer *thePort*, but you may prefer to use *GetPort* for better readability of your program text. For

example, a procedure could do a *GetPort (savePort)* before setting its own *grafPort* and a *SetPort (savePort)* afterwards to restore the previous port.

```
procedure GrafDevice( device : integer );
```

GrafDevice sets *thePort^.device* to the given number, which identifies the logical output device for this *grafPort*. *QuickDraw* uses this information. The initial device number is 0, which represents the Macintosh screen.

```
procedure SetPortBits( bm : BitMap );
```

SetPortBits sets *thePort^.portBits* to any previously defined bitmap. This allows you to perform all normal drawing and calculations on a buffer other than the Macintosh screen--for example, a 640-by-7 output buffer for a dot matrix printer, or a small off-screen image for later "stamping" onto the screen.

Remember to prepare all fields of the bitmap before you call *SetPortBits*.

```
procedure PortSize( width, height : integer );
```

PortSize changes the size of the current *grafPort*'s *portRect*. *This does not affect the screen*; it merely changes the size of the "active area" of the *grafPort*.

The top left corner of the *portRect* remains at its same location; the width and height of the *portRect* are set to the given width and height. In other words, *PortSize* moves the bottom right corner of the *portRect* to a position relative to the top left corner.

PortSize does not change the *clipRgn* or the *visRgn*, nor does it affect the local coordinate system of the *grafPort*: it changes only the *portRect*'s width and height. Remember that all drawing occurs only in the intersection of the *portBits.bounds* and the *portRect*, clipped to the *visRgn* and the *clipRgn*.

```
procedure MovePortTo( leftGlobal, topGlobal : integer );
```

MovePortTo changes the position of the current *grafPort*'s *portRect*. *This does not affect the screen*; it merely changes the location at which subsequent drawing inside the port will appear.

The *leftGlobal* and *topGlobal* parameters set the distance between the top left corner of the *portBits.bounds* and the top left corner of the new *portRect*. For example,

```
MovePortTo(256,171);
```

will move the top left corner of the *portRect* to the center of the screen (if *portBits* is the Macintosh screen) regardless of the local coordinate system.

Like `PortSize`, `MovePortTo` does not change the `clipRgn` or the `visRgn`, nor does it affect the local coordinate system of the `grafPort`.

procedure `SetOrigin(h, v : integer);`

`SetOrigin` changes the local coordinate system of the current `grafPort`. *This does not affect the screen*; it does, however, affect where subsequent drawing and calculation will appear in the `grafPort`. `SetOrigin` updates the coordinates of the `portBits.bounds`, the `portRect`, and the `visRgn`. All subsequent drawing and calculation routines will use the new coordinate system.

The `h` and `v` parameters set the coordinates of the top left corner of the `portRect`. All other coordinates are calculated from this point. All relative distances among any elements in the port will remain the same; only their absolute local coordinates will change.

Note: `SetOrigin` does not update the coordinates of the `clipRgn` or the pen; these items stick to the coordinate system (unlike the port's structure, which sticks to the screen).

`SetOrigin` is useful for adjusting the coordinate system after a scrolling operation. (See `ScrollRect` in Section C.8.14, Bit Transfer Operations.)

procedure `SetClip(rgn : RgnHandle);`

`SetClip` changes the clipping region of the current `grafPort` to a region equivalent to the given region. Note that this does not change the region handle, but affects the clipping region itself. Since `SetClip` makes a copy of the given region, any subsequent changes you make to that region will not affect the clipping region of the port.

You can set the clipping region to any arbitrary region, to aid you in drawing inside the `grafPort`. The initial `clipRgn` is an arbitrarily large rectangle.

procedure `GetClip(rgn : RgnHandle);`

`GetClip` changes the given region to a region equivalent to the clipping region of the current `grafPort`. This is the reverse of what `SetClip` does. Like `SetClip`, it does not change the region handle.

procedure `ClipRect(r : Rect);`

`ClipRect` changes the clipping region of the current `grafPort` to a rectangle equivalent to given rectangle. Note that this does not change the region handle, but affects the region itself.

procedure `BackPat(pat : Pattern);`

BackPat sets the background pattern of the current grafPort to the given pattern. The background pattern is used in ScrollRect and in all QuickDraw routines that perform an "erase" operation.

C.8.2 Cursor-Handling Routines

procedure *InitCursor*;

InitCursor sets the current cursor to the predefined *arrow* cursor, an arrow pointing north-northwest, and sets the *cursor level* to 0, making the cursor visible. The cursor level, which is initialized to 0 when the system is booted, keeps track of the number of times the cursor has been hidden to compensate for nested calls to HideCursor and ShowCursor (below).

Before you call InitCursor, the cursor is undefined (or, if set by a previous process, it's whatever that process set it to).

procedure *SetCursor*(*crsr* : *Cursor*);

SetCursor sets the current cursor to the 16-by-16-bit image in *crsr*. If the cursor is hidden, it remains hidden and will attain the new appearance when it's uncovered; if the cursor is already visible, it changes to the new appearance immediately.

The cursor image is initialized by InitCursor to a north-northwest arrow, visible on the screen. There is no way to retrieve the current cursor image.

procedure *HideCursor*;

HideCursor removes the cursor from the screen, restoring the bits under it, and decrements the cursor level (which InitCursor initialized to 0). Every call to HideCursor should be balanced by a subsequent call to ShowCursor.

procedure *ShowCursor*;

ShowCursor increments the cursor level, which may have been decremented by HideCursor, and displays the cursor on the screen if the level becomes 0. A call to ShowCursor should balance each previous call to HideCursor. The level is not incremented beyond 0, so extra calls to ShowCursor don't hurt.

If the cursor has been changed (with SetCursor) while hidden, ShowCursor presents the new cursor.

The cursor is initialized by InitCursor to a north-northwest arrow, not hidden.

procedure *ObscureCursor*;

ObscureCursor hides the cursor until the next time the mouse is moved. Unlike *HideCursor*, it has no effect on the cursor level and must not be balanced by a call to *ShowCursor*.

C.8.3 Pen and Line-Drawing Routines

The pen and line-drawing routines all depend on the coordinate system of the current *grafPort*. Remember that each *grafPort* has its own pen; if you draw in one *grafPort*, change to another, and return to the first, the pen will have remained in the same location.

procedure *HidePen*;

HidePen decrements the current *grafPort*'s *pnVis* field, which is initialized to 0 by *OpenPort*; whenever *pnVis* is negative, the pen does not draw on the screen. *pnVis* keeps track of the number of times the pen has been hidden to compensate for nested calls to *HidePen* and *ShowPen* (below). *HidePen* is called by *OpenRgn*, *OpenPicture*, and *OpenPoly* so that you can define regions, pictures, and polygons without drawing on the screen.

procedure *ShowPen*;

ShowPen increments the current *grafPort*'s *pnVis* field, which may have been decremented by *HidePen*; if *pnVis* becomes 0, *QuickDraw* resumes drawing on the screen. Extra calls to *ShowPen* will increment *pnVis* beyond 0, so every call to *ShowPen* should be balanced by a subsequent call to *HidePen*. *ShowPen* is called by *CloseRgn*, *ClosePicture*, and *ClosePoly*.

procedure *GetPen*(**var** *pt* : *Point*);

GetPen returns the current pen location, in the local coordinates of the current *grafPort*.

procedure *GetPenState*(**var** *pnState* : *PenState*);

GetPenState saves the pen location, size, pattern, and mode in a storage variable, to be restored later with *SetPenState* (below). This is useful when calling short subroutines that operate in the current port but must change the graphics pen: each such procedure can save the pen's state when it's called, do whatever it needs to do, and restore the previous pen state immediately before returning.

The *PenState* data type is not useful for anything except saving the pen's state.

```
procedure SetPenState( pnState : PenState );
```

SetPenState sets the pen location, size, pattern, and mode in the current grafPort to the values stored in *pnState*. This is usually called at the end of a procedure that has altered the pen parameters and wants to restore them to their state at the beginning of the procedure. (See GetPenState, above.)

```
procedure PenSize( width, height : integer );
```

PenSize sets the dimensions of the graphics pen in the current grafPort. All subsequent calls to Line, LineTo, and the procedures that draw framed shapes in the current grafPort will use the new pen dimensions.

The pen dimensions can be accessed in the variable *thePort^.pnSize*, which is of type *Point*. If either of the pen dimensions is set to a negative value, the pen assumes the dimensions (0,0) and no drawing is performed. For a discussion of how the pen draws, see Section C.6, General Discussion of Drawing.

```
procedure PenMode( mode : integer );
```

PenMode sets the transfer mode through which the *pnPat* is transferred onto the bitmap when lines or shapes are drawn. The mode may be any one of the pattern transfer modes:

<i>patCopy</i>	<i>patXor</i>	<i>notPatCopy</i>	<i>notPatXor</i>
<i>patOr</i>	<i>patBic</i>	<i>notPatOr</i>	<i>notPatBic</i>

If the mode is one of the source transfer modes (or negative), no drawing is performed. The current pen mode can be obtained in the variable *thePort^.pnMode*. The initial pen mode is *patCopy*, in which the pen pattern is copied directly to the bitmap.

```
procedure PenPat( pat : Pattern );
```

PenPat sets the pattern that is used by the pen in the current grafPort. The standard patterns *white*, *black*, *gray*, *ltGray*, and *dkGray* are predefined; the initial pen pattern is *black*. The current pen pattern can be obtained in the variable *thePort^.pnPat*, and this value can be assigned (but not compared!) to any other variable of type *Pattern*.

```
procedure PenNormal;
```

PenNormal resets the initial state of the pen in the current grafPort, as follows:

Field	Setting
<i>pnSize</i>	<i>(1,1)</i>
<i>pnMode</i>	<i>patCopy</i>
<i>pnPat</i>	<i>black</i>

The pen location is not changed.

procedure *MoveTo*(*h, v : integer*);

MoveTo moves the pen to location (h, v) in the local coordinates of the current *grafPort*. No drawing is performed.

procedure *Move*(*dh, dv : integer*);

Move moves the pen a distance of *dh* horizontally and *dv* vertically from its current location; it calls *MoveTo* $(h+dh, v+dv)$, where (h, v) is the current location. The positive directions are to the right and down. No drawing is performed.

procedure *LineTo*(*h, v : integer*);

LineTo draws a line from the current pen location to the location specified (in local coordinates) by *h* and *v*. The new pen location is (h, v) after the line is drawn. See Section C.6, General Discussion of Drawing.

If a region or polygon is open and being formed, its outline is infinitely thin and is not affected by the *pnSize*, *pnMode*, or *pnPat*. (See *OpenRgn* and *OpenPoly*.)

procedure *Line*(*dh, dv : integer*);

Line draws a line to the location that is a distance of *dh* horizontally and *dv* vertically from the current pen location; it calls *LineTo* $(h+dh, v+dv)$, where (h, v) is the current location. The positive directions are to the right and down. The pen location becomes the coordinates of the end of the line after the line is drawn. See Section C.6, General Discussion of Drawing.

If a region or polygon is open and being formed, its outline is infinitely thin and is not affected by the *pnSize*, *pnMode*, or *pnPat*. (See *OpenRgn* and *OpenPoly*.)

C.8.4 Text-Drawing Routines

Each `grafPort` has its own text characteristics, and all these procedures deal with those of the current port.

procedure *TextFont*(*font* : integer);

TextFont sets the current `grafPort`'s font (*thePort*^.*txFont*) to the given font number. The initial font number is 0, which represents the system font.

procedure *TextFace*(*face* : Style);

TextFace sets the current `grafPort`'s character style (*thePort*^.*txFace*). The *Style* data type allows you to specify a set of one or more of the following predefined constants: *bold*, *italic*, *underline*, *outline*, *shadow*, *condense*, and *extend*. For example:

<i>TextFace</i> ([<i>bold</i>]);	{ <i>bold</i> }
<i>TextFace</i> ([<i>bold</i> , <i>italic</i>]);	{ <i>bold</i> and <i>italic</i> }
<i>TextFace</i> (<i>thePort</i> ^. <i>txFace</i> + [<i>bold</i>]);	{whatever it was plus <i>bold</i> }
<i>TextFace</i> (<i>thePort</i> ^. <i>txFace</i> - [<i>bold</i>]);	{whatever it was but no <i>bold</i> }
<i>TextFace</i> ([]);	{ <i>normal</i> }

procedure *TextMode*(*mode* : integer);

TextMode sets the current `grafPort`'s transfer mode for drawing text (*thePort*^.*txMode*).

The mode should be *srcOr*, *srcXor*, or *srcBiC*. The initial transfer mode for drawing text is *srcOr*.

procedure *TextSize*(*size* : integer);

TextSize sets the current `grafPort`'s type size (*thePort*^.*txSize*) to the given number of points. Any size may be specified, but the result will look best if QuickDraw has the font in that size (otherwise it will scale a size it does have). The next best result will occur if the given size is an even multiple of a size available for the font. If 0 is specified, QuickDraw will choose the system font size, 12 point, scaling to that size if necessary. The initial *txSize* setting is 0.

procedure *SpaceExtra*(*extra* : longint);

SpaceExtra sets the current `grafPort`'s *spExtra* field, which specifies the average number of pixels (a fixed-point number - see Fixed Point Math in Appendix E) by which to widen each space in a line of text. This is useful when text is being fully justified (that is, aligned with both a left and a right margin). Consider, for example, a line that contains three spaces; if there would normally be six pixels between the end of the line and the right margin, you would call

SpaceExtra (*FixRatio* (6, 3)) to print the line with full justification. The initial *spExtra* setting is 0.

Note: *SpaceExtra* will also take a negative argument, but be careful not to narrow spaces so much that the text is unreadable.

procedure *DrawChar* (*ch* : *char*);

DrawChar places the given character to the right of the pen location, with the left end of its base line at the pen's location, and advances the pen accordingly. If the character is not in the font, the font's missing symbol is drawn.

procedure *DrawString* (*s* : *Str255*);

DrawString performs consecutive calls to *DrawChar* for each character in the supplied string; the string is placed beginning at the current pen location and extending right. No formatting (carriage returns, line feeds, etc.) is performed by *QuickDraw*. The pen location ends up to the right of the last character in the string.

procedure *DrawText* (*textBuf* : *Ptr*;
 firstByte, *byteCount* : *integer*);

DrawText draws text from an arbitrary structure in memory specified by *textBuf*, starting *firstByte* bytes into the structure and continuing for *byteCount* bytes. The string of text is placed beginning at the current pen location and extending right. No formatting (carriage returns, line feeds, etc.) is performed by *QuickDraw*. The pen location ends up to the right of the last character in the string. *ByteCount* must be in the range 1..255. *FirstByte* should be zero to start at the first character in *textBuf*.

function *CharWidth* (*ch* : *char*) : *integer*;

CharWidth returns the value that will be added to the pen horizontal coordinate if the specified character is drawn. *CharWidth* includes the effects of the stylistic variations set with *TextFace*; if you change these after determining the character width but before actually drawing the character, the predetermined width may not be correct. If the character is a space, *CharWidth* also includes the effect of *SpaceExtra*.

function *StringWidth* (*s* : *Str255*) : *integer*;

StringWidth returns the width of the given text string, which it calculates by adding the widths of all the characters in the string (see *CharWidth*, above). This value will be added to the pen horizontal coordinate if the specified string is drawn.

```
function TextWidth( textBuf: Ptr;
                    firstByte, byteCount: integer ) : integer;
```

TextWidth returns the width of the text stored in the arbitrary structure in memory specified by *textBuf*, starting *firstByte* bytes into the structure and continuing for *byteCount* bytes. It calculates the width by adding the widths of all the characters in the text. (See CharWidth, above.)

```
procedure GetFontInfo( var info : FontInfo );
```

GetFontInfo returns the following information about the current grafPort's character font, taking into consideration the style and size in which the characters will be drawn: the ascent, descent, maximum character width (the greatest distance the pen will move when a character is drawn), and leading (the vertical distance between the descent line and the ascent line below it), all in pixels. The *FontInfo* data structure is defined as:

```
type
    FontInfo = record
        ascent: integer;
        descent: integer;
        widMax: integer;
        leading: integer
    end;
```

C.8.5 Drawing in Color

These routines will enable applications to do color drawing in the future when Apple supports color output devices for the Macintosh. All nonwhite colors will appear as black on black-and-white output devices.

```
procedure ForeColor( color : longint );
```

ForeColor sets the foreground color for all drawing in the current grafPort (*thePort^.fgColor*) to the given color. The following standard colors are predefined: *blackColor*, *whiteColor*, *redColor*, *greenColor*, *blueColor*, *cyanColor*, *magentaColor*, and *yellowColor*. The initial foreground color is *blackColor*.

```
procedure BackColor( color : longint );
```

BackColor sets the background color for all drawing in the current grafPort (*thePort^.bkColor*) to the given color. Eight standard colors are predefined (see ForeColor, above). The initial background color is *whiteColor*.

```
procedure ColorBit( whichBit : integer );
```

ColorBit is called by printing software for a color printer, or other color- imaging software, to set the current grafPort's *colrBit* field to *whichBit*; this tells QuickDraw which plane of the color picture to draw into. QuickDraw will draw into the plane corresponding to bit number *whichBit*. Since QuickDraw can support output devices that have up to 32 bits of color information per pixel, the possible range of values for *whichBit* is 0 through 31. The initial value of the *colrBit* field is 0.

C.8.6 Calculations with Points

```
procedure AddPt( srcPt : Point; var dstPt : Point );
```

AddPt adds the coordinates of *srcPt* to the coordinates of *dstPt*, and returns the result in *dstPt*.

```
procedure SubPt( srcPt : Point; var dstPt : Point );
```

SubPt subtracts the coordinates of *srcPt* from the coordinates of *dstPt*, and returns the result in *dstPt*.

```
procedure SetPt( var pt : Point; h, v : integer );
```

SetPt assigns two integer coordinates to a variable of type *Point*.

```
function EqualPt( ptA, ptB : Point ) : boolean;
```

EqualPt compares the two points and returns *true* if they are equal or *false* if not.

```
procedure ScalePt( var pt : Point; srcRect, dstRect : Rect );
```

A width and height are passed in *pt*; the horizontal component of *pt* is the width, and the vertical component of *pt* is the height. ScalePt scales these measurements as follows and returns the result in *pt*: it multiplies the given width by the ratio of *dstRect*'s width to *srcRect*'s width, and multiplies the given height by the ratio of *dstRect*'s height to *srcRect*'s height. In Figure C-18, where *dstRect*'s width is twice *srcRect*'s width and its height is three times *srcRect*'s height, the pen width is scaled from 3 to 6 and the pen height is scaled from 2 to 6.

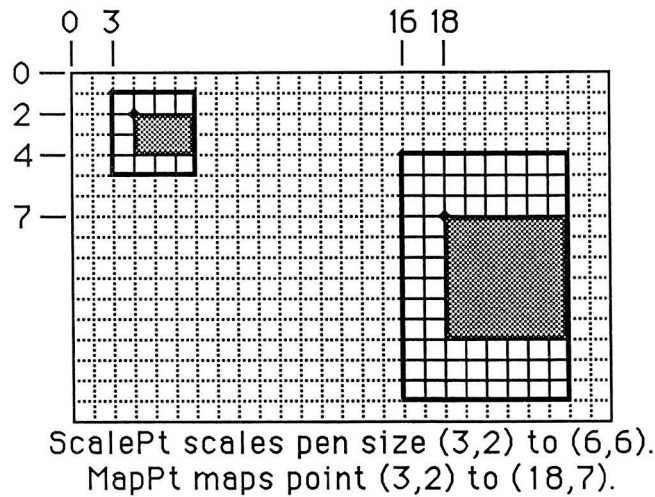


Figure C-18
ScalePt and MapPt

```
procedure MapPt( var pt : Point; srcRect, dstRect : Rect );
```

Given a point within *srcRect*, MapPt maps it to a similarly located point within *dstRect* (that is, to where it would fall if it were part of a drawing being expanded or shrunk to fit *dstRect*). The result is returned in *pt*. A corner point of *srcRect* would be mapped to the corresponding corner point of *dstRect*, and the center of *srcRect* to the center of *dstRect*. In Figure C-18 above, the point (3,2) in *srcRect* is mapped to (18,7) in *dstRect*. *FromRect* and *dstRect* may overlap, and *pt* need not actually be within *srcRect*.

Warning: Remember, if you are going to draw inside the rectangle in *dstRect*, you will probably also want to scale the pen size accordingly with ScalePt.

```
procedure LocalToGlobal( var pt : Point );
```

LocalToGlobal converts the given point from the current grafPort's local coordinate system into a global coordinate system with the origin (0,0) at the top left corner of the port's bit image (such as the screen). This global point can then be compared to other global points, or be changed into the local coordinates of another grafPort.

Since a rectangle is defined by two points, you can convert a rectangle into global coordinates by performing two LocalToGlobal calls. You can also convert a rectangle, region, or polygon into global coordinates by calling OffsetRect, OffsetRgn, or OffsetPoly. For examples, see GlobalToLocal below.

```
procedure GlobalToLocal( var pt : Point );
```

GlobalToLocal takes a point expressed in global coordinates (with the top left corner of the bitmap as coordinate (0,0)) and converts it into the local coordinates of the current grafPort. The global point can be obtained with the LocalToGlobal call (see above). For example, suppose a

game draws a "ball" within a rectangle named *ballRect*, defined in the grafPort named *gamePort* (as illustrated below in Figure C-19). If you want to draw that ball in the grafPort named *selectPort*, you can calculate the ball's *selectPort* coordinates like this:

```
SetPort(gamePort);           { start in origin port  }
selectBall := ballRect;      { make a copy to be moved }
LocalToGlobal(selectBall.topLeft); { put both corners into }
LocalToGlobal(selectBall.botRight); { global coordinates }
SetPort(selectPort);         { switch to dest. port }
GlobalToLocal(selectBall.topLeft); { put both corners into }
GlobalToLocal(selectBall.botRight); { these local coordinates }
FillOval(selectBall,ballColor); { now you have the ball! }
```

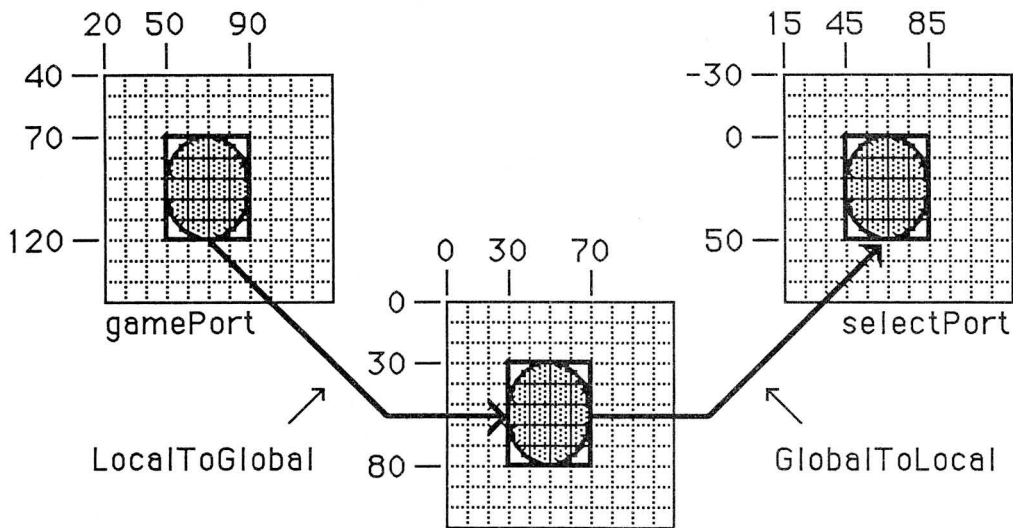


Figure C-19
Converting between Coordinate Systems

You can see from Figure C-19 that *LocalToGlobal* and *GlobalToLocal* simply offset the coordinates of the rectangle by the coordinates of the top left corner of the local grafPort's boundary rectangle. You could also do this with *OffsetRect*. In fact, the way to convert regions and polygons from one coordinate system to another is with *OffsetRgn* or *OffsetPoly* rather than *LocalToGlobal* and *GlobalToLocal*. For example, if *myRgn* were a region enclosed by a rectangle having the same coordinates as *ballRect* in *gamePort*, you could convert the region to global coordinates with

```
OffsetRgn(myRgn, -20, -40);
```

and then convert it to the coordinates of the *selectPort* grafPort with

```
OffsetRgn(myRgn, 15, -30);
```

C.8.7 Calculations with Rectangles

Calculation routines are independent of the current coordinate system; a calculation will operate the same regardless of which grafPort is active.

Note: Remember that if the parameters to one of the calculation routines were defined in different grafPorts, you must first adjust them to be in the same coordinate system. If you do not adjust them, the result returned by the routine may be different from what you see on the screen. To adjust to a common coordinate system, see LocalToGlobal and GlobalToLocal in Section C.8.6, Calculations with Points.

```
procedure SetRect( var r : Rect;  
                  left, top, right, bottom : integer )
```

SetRect assigns the four boundary coordinates to the rectangle. The result is a rectangle with coordinates (left,top,right,bottom).

This procedure is supplied as a utility to help you shorten your program text. If you want a more readable text at the expense of length, you can assign integers (or points) directly into the rectangle's fields. There is no significant code size or execution speed advantage to either method; one's just easier to write, and the other's easier to read.

```
procedure OffsetRect( var r : Rect; dh, dv : integer );
```

OffsetRect moves the rectangle by adding *dh* to each horizontal coordinate and *dv* to each vertical coordinate. If *dh* and *dv* are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The rectangle retains its shape and size; it's merely moved on the coordinate plane. This does not affect the screen unless you subsequently call a routine to draw within the rectangle.

```
procedure MapRect( var r : Rect; srcRect, dstRect : Rect );
```

Given a rectangle within *srcRect*, MapRect maps it to a similarly located rectangle within *dstRect* by calling MapPt to map the top left and bottom right corners of the rectangle. The result is returned in *r*.

```
procedure InsetRect( var r : Rect; dh, dv : integer );
```

InsetRect shrinks or expands the rectangle. The left and right sides are moved in by the amount specified by *dh*; the top and bottom are moved toward the center by the amount specified by *dv*. If *dh* or *dv* is negative, the appropriate pair of sides is moved outward instead of inward. The effect is to alter the size by $2 * dh$ horizontally and $2 * dv$ vertically, with the rectangle remaining centered in the same place on the coordinate plane.

If the resulting width or height becomes less than 1, the rectangle is set to the empty rectangle (0,0,0,0).

```
function SectRect( srcRectA, srcRectB : Rect;  
                  var dstRect : Rect ) : boolean;
```

SectRect calculates the rectangle that is the intersection of the two input rectangles, and returns *true* if they indeed intersect or *false* if they do not. Rectangles that "touch" at a line or a point are not considered intersecting, because their intersection rectangle (really, in this case, an intersection line or point) does not enclose any bits on the bitmap.

If the rectangles do not intersect, the destination rectangle is set to (0,0,0,0). SectRect works correctly even if one of the source rectangles is also the destination.

```
procedure UnionRect( srcRectA, srcRectB : Rect;  
                    var dstRect : Rect );
```

UnionRect calculates the smallest rectangle which encloses both input rectangles. It works correctly even if one of the source rectangles is also the destination.

```
function PtInRect( pt : Point; r : Rect ) : boolean;
```

PtInRect determines whether the pixel below and to the right of the given coordinate point is enclosed in the specified rectangle, and returns *true* if so or *false* if not.

```
procedure Pt2Rect( ptA, ptB : Point; var dstRect : Rect );
```

Pt2Rect returns the smallest rectangle which encloses the two input points.

```
procedure PtToAngle( r : Rect; pt : Point;  
                   var angle : integer );
```

PtToAngle calculates an integer angle between a line from the center of the rectangle to the given point and a line from the center of the rectangle pointing straight up (12 o'clock high). The angle is in degrees from 0 to 359, measured clockwise from 12 o'clock, with 90° at 3 o'clock, 180° at 6 o'clock, and 270° at 9 o'clock. Other angles are measured relative to the rectangle: If the line to the given point goes through the top right corner of the rectangle, the angle returned is 45 degrees, even if the rectangle is not square; if it goes through the bottom right corner, the angle is 135 degrees, and so on (see Figure C-20).

The angle returned might be used as input to one of the procedures that manipulate arcs and wedges, as described in Section C.8.13, Graphic Operations on Arcs and Wedges.

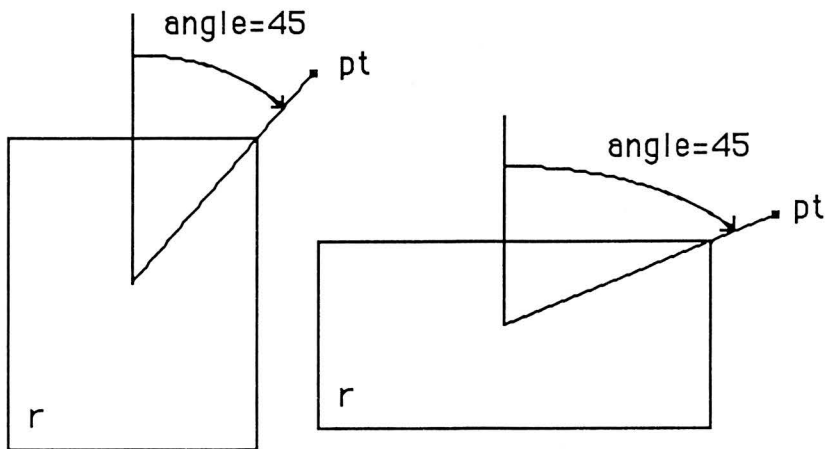


Figure C-20
PtToAngle

```
function EqualRect( rectA, rectB : Rect ) : boolean;
```

EqualRect compares the two rectangles and returns *true* if they are equal or *false* if not. The two rectangles must have identical boundary coordinates to be considered equal.

```
function EmptyRect( r : Rect ) : boolean;
```

EmptyRect returns *true* if the given rectangle is an empty rectangle or *false* if not. A rectangle is considered empty if the bottom coordinate is equal to or less than the top or the right coordinate is equal to or less than the left.

C.8.8 Graphic Operations on Rectangles

These procedures perform graphic operations on rectangles. See also ScrollRect in Section C.8.14, Bit Transfer Operations.

```
procedure FrameRect( r : Rect );
```

FrameRect draws an outline just inside the specified rectangle, using the current grafPort's pen pattern, mode, and size. The outline is as wide as the pen width and as tall as the pen height. It is drawn with the *pnPat*, according to the pattern transfer mode specified by *pnMode*. The pen location is not changed by this procedure.

If a region is open and being formed, the outside outline of the new rectangle is mathematically added to the region's boundary.

procedure *PaintRect*(*r* : *Rect*);

PaintRect paints the specified rectangle with the current *grafPort*'s pen pattern and mode. The rectangle on the bitmap is filled with the *pnPat*, according to the pattern transfer mode specified by *pnMode*. The pen location is not changed by this procedure.

procedure *EraseRect*(*r* : *Rect*);

EraseRect paints the specified rectangle with the current *grafPort*'s back- ground pattern *bkPat* (in *patCopy* mode). The *grafPort*'s *pnPat* and *pnMode* are ignored; the pen location is not changed.

procedure *InvertRect*(*r* : *Rect*);

InvertRect inverts the pixels enclosed by the specified rectangle: every white pixel becomes black and every black pixel becomes white. The *grafPort*'s *pnPat*, *pnMode*, and *bkPat* are all ignored; the pen location is not changed.

procedure *FillRect*(*r* : *Rect*; *pat* : *Pattern*);

FillRect fills the specified rectangle with the given pattern (in *patCopy* mode). The *grafPort*'s *pnPat*, *pnMode*, and *bkPat* are all ignored; the pen location is not changed.

C.8.9 Graphic Operations on Ovals

Ovals are drawn inside rectangles that you specify. If the rectangle you specify is square, *QuickDraw* draws a circle.

procedure *FrameOval*(*r* : *Rect*);

FrameOval draws an outline just inside the oval that fits inside the specified rectangle, using the current *grafPort*'s pen pattern, mode, and size. The outline is as wide as the pen width and as tall as the pen height. It is drawn with the *pnPat*, according to the pattern transfer mode specified by *pnMode*. The pen location is not changed by this procedure.

If a region is open and being formed, the outside outline of the new oval is mathematically added to the region's boundary.

procedure *PaintOval*(*r* : *Rect*);

PaintOval paints an oval just inside the specified rectangle with the current *grafPort*'s pen pattern and mode. The oval on the bitmap is filled with the *pnPat*, according to the pattern transfer mode specified by *pnMode*. The pen location is not changed by this procedure.

procedure *EraseOval*(*r* : *Rect*);

EraseOval paints an oval just inside the specified rectangle with the current *grafPort*'s background pattern *bkPat* (in *patCopy* mode). The *grafPort*'s *pnPat* and *pnMode* are ignored; the pen location is not changed.

procedure *InvertOval*(*r* : *Rect*);

InvertOval inverts the pixels enclosed by an oval just inside the specified rectangle: every white pixel becomes black and every black pixel becomes white. The *grafPort*'s *pnPat*, *pnMode*, and *bkPat* are all ignored; the pen location is not changed.

procedure *FillOval*(*r* : *Rect*; *pat* : *Pattern*);

FillOval fills an oval just inside the specified rectangle with the given pattern (in *patCopy* mode). The *grafPort*'s *pnPat*, *pnMode*, and *bkPat* are all ignored; the pen location is not changed.

C.8.10 Graphic Operations on Rounded-Corner Rectangles

procedure *FrameRoundRect*(*r* : *Rect*;
 ovalWidth, *ovalHeight* : *integer*);

FrameRoundRect draws an outline just inside the specified rounded-corner rectangle, using the current *grafPort*'s pen pattern, mode, and size. *OvalWidth* and *ovalHeight* specify the diameters of curvature for the corners (see Figure C-21). The outline is as wide as the pen width and as tall as the pen height. It is drawn with the *pnPat*, according to the pattern transfer mode specified by *pnMode*. The pen location is not changed by this procedure.

If a region is open and being formed, the outside outline of the new rounded-corner rectangle is mathematically added to the region's boundary.

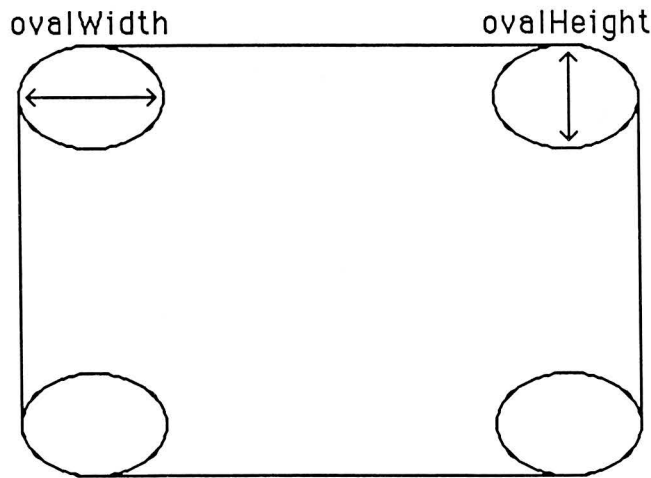


Figure C-21
Rounded-Corner Rectangle

```
procedure PaintRoundRect( r : Rect;
                           ovalWidth, ovalHeight : integer );
```

PaintRoundRect paints the specified rounded-corner rectangle with the current grafPort's pen pattern and mode. *OvalWidth* and *ovalHeight* specify the diameters of curvature for the corners. The rounded-corner rectangle on the bitmap is filled with the *pnPat*, according to the pattern transfer mode specified by *pnMode*. The pen location is not changed by this procedure.

```
procedure EraseRoundRect( r : Rect;
                           ovalWidth, ovalHeight : integer );
```

EraseRoundRect paints the specified rounded-corner rectangle with the current grafPort's background pattern *bkPat* (in *patCopy* mode). *OvalWidth* and *ovalHeight* specify the diameters of curvature for the corners. The grafPort's *pnPat* and *pnMode* are ignored; the pen location is not changed.

```
procedure InvertRoundRect( r : Rect;
                            ovalWidth, ovalHeight : integer );
```

InvertRoundRect inverts the pixels enclosed by the specified rounded-corner rectangle: every white pixel becomes black and every black pixel becomes white. *OvalWidth* and *ovalHeight* specify the diameters of curvature for the corners. The grafPort's *pnPat*, *pnMode*, and *bkPat* are all ignored; the pen location is not changed.

```
procedure FillRoundRect( r : Rect;
                        ovalWidth, ovalHeight : integer;
                        pat : Pattern );
```

FillRoundRect fills the specified rounded-corner rectangle with the given pattern (in *patCopy* mode). *OvalWidth* and *ovalHeight* specify the diameters of curvature for the corners. The grafPort's *pnPat*, *pnMode*, and *bkPat* are all ignored; the pen location is not changed.

C.8.11 Graphic Operations on Arcs and Wedges

These procedures perform graphic operations on arcs and wedge-shaped sections of ovals. See also PtToAngle in Section C.8.7, Calculations with Rectangles.

```
procedure FrameArc( r : Rect; startAngle, arcAngle : integer );
```

FrameArc draws an arc of the oval that fits inside the specified rectangle, using the current grafPort's pen pattern, mode, and size. *StartAngle* indicates where the arc begins and is treated mod 360. *ArcAngle* defines the extent of the arc. The angles are given in positive or negative degrees; a positive angle goes clockwise, while a negative angle goes counterclockwise. Zero degrees is at 12 o'clock high, 90° (or -270°) is at 3 o'clock, 180° (or -180°) is at 6 o'clock, and 270° (or -90°) is at 9 o'clock. Other angles are measured relative to the enclosing rectangle: a line from the center of the rectangle through its top right corner is at 45 degrees, even if the rectangle is not square; a line through the bottom right corner is at 135 degrees, and so on (see Figure C-22).

The arc is as wide as the pen width and as tall as the pen height. It is drawn with the *pnPat*, according to the pattern transfer mode specified by *pnMode*. The pen location is not changed by this procedure.

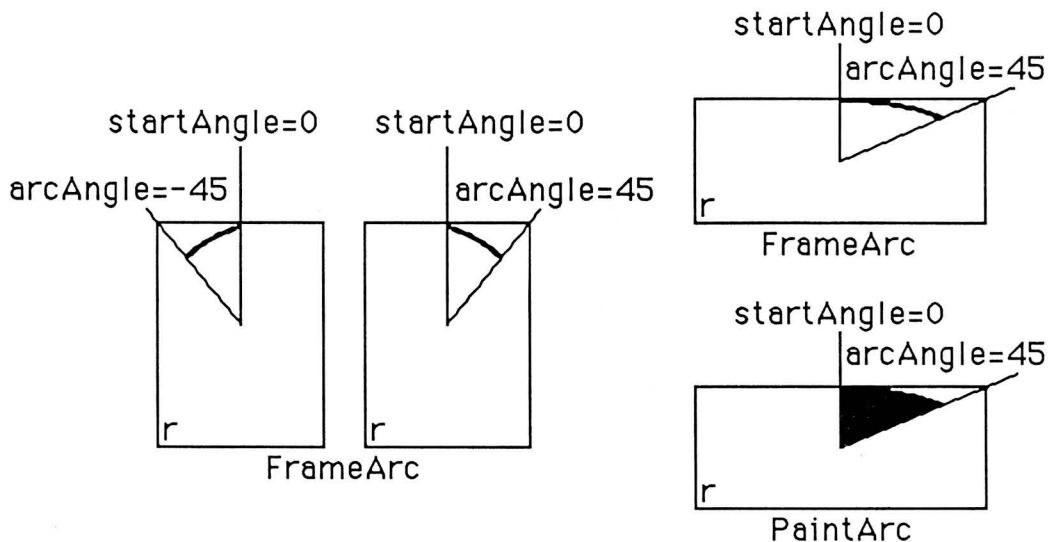


Figure C-22
Operations on Arcs and Wedges

Warning: FrameArc differs from other QuickDraw procedures that frame shapes in that the arc is not mathematically added to the boundary of a region that is open and being formed.

procedure PaintArc(*r* : Rect; *startAngle*, *arcAngle* : integer);

PaintArc paints a wedge of the oval just inside the specified rectangle with the current grafPort's pen pattern and mode. *StartAngle* and *arcAngle* define the arc of the wedge as in FrameArc. The wedge on the bitmap is filled with the *pnPat*, according to the pattern transfer mode specified by *pnMode*. The pen location is not changed by this procedure.

procedure EraseArc(*r* : Rect; *startAngle*, *arcAngle* : integer);

EraseArc paints a wedge of the oval just inside the specified rectangle with the current grafPort's background pattern *bkPat* (in *patCopy* mode). *StartAngle* and *arcAngle* define the arc of the wedge as in FrameArc. The grafPort's *pnPat* and *pnMode* are ignored; the pen location is not changed.

procedure InvertArc(*r* : Rect; *startAngle*, *arcAngle* : integer);

InvertArc inverts the pixels enclosed by a wedge of the oval just inside the specified rectangle: every white pixel becomes black and every black pixel becomes white. *StartAngle* and *arcAngle* define the arc of the wedge as in FrameArc. The grafPort's *pnPat*, *pnMode*, and *bkPat* are all ignored; the pen location is not changed.

procedure FillArc(*r* : Rect; *startAngle*, *arcAngle* : integer;
 pat : Pattern);

FillArc fills a wedge of the oval just inside the specified rectangle with the given pattern (in *patCopy* mode). *StartAngle* and *arcAngle* define the arc of the wedge as in FrameArc. The grafPort's *pnPat*, *pnMode*, and *bkPat* are all ignored; the pen location is not changed.

C.8.12 Calculations with Regions

Note: Remember that if the parameters to one of the calculation routines were defined in different grafPorts, you must first adjust them to be in the same coordinate system. If you do not adjust them, the result returned by the routine may be different from what you see on the screen. To adjust to a common coordinate system, see LocalToGlobal and GlobalToLocal in Section C.8.6, Calculations with Points.

function NewRgn : RgnHandle;

NewRgn allocates space for a new, dynamic, variable-size region, initializes it to the empty region (0,0,0,0), and returns a handle to the new region. Only this function creates new regions; all other procedures just alter the size and shape of regions you create. OpenPort calls NewRgn to allocate space for the port's *visRgn* and *clipRgn*.

Warnings: Except when using *visRgn* or *clipRgn*, you *must* call *NewRgn* before specifying a region's handle in any drawing or calculation procedure.

Never refer to a region without using its handle.

procedure *DisposeRgn*(*rgn* : *RgnHandle*);

DisposeRgn deallocates space for the region whose handle is supplied, and returns the memory used by the region to the free memory pool. Use this only after you are completely through with a temporary region.

Warning: Never use a region once you have deallocated it, or you will risk being hung by dangling pointers!

procedure *CopyRgn*(*srcRgn*, *dstRgn* : *RgnHandle*);

CopyRgn copies the mathematical structure of *srcRgn* into *dstRgn*; that is, it makes a duplicate copy of *srcRgn*. Once this is done, *srcRgn* may be altered (or even disposed of) without affecting *dstRgn*. *CopyRgn does not create the destination region*: you must use *NewRgn* to create the *dstRgn* before you call *CopyRgn*.

procedure *SetEmptyRgn*(*rgn* : *RgnHandle*);

SetEmptyRgn destroys the previous structure of the given region, then sets the new structure to the empty region (0,0,0,0).

procedure *SetRectRgn*(*rgn* : *RgnHandle*;
 left, *top*, *right*, *bottom* : *integer*);

SetRectRgn destroys the previous structure of the given region, then sets the new structure to the rectangle specified by *left*, *top*, *right*, and *bottom*.

If the specified rectangle is empty (i.e., *left* >= *right* or *top* >= *bottom*), the region is set to the empty region (0,0,0,0).

procedure *RectRgn*(*rgn* : *RgnHandle*; *r* : *Rect*);

RectRgn destroys the previous structure of the given region, then sets the new structure to the rectangle specified by *r*. This is operationally synonymous with *SetRectRgn*, except the input rectangle is defined by a rectangle rather than by four boundary coordinates.

procedure *OpenRgn*;

OpenRgn tells QuickDraw to allocate temporary space and start saving lines and framed shapes for later processing as a region definition. While a region is open, all calls to *Line*, *LineTo*, and

the procedures that draw framed shapes (except arcs) affect the outline of the region. Only the line endpoints and shape boundaries affect the region definition; the pen mode, pattern, and size do not affect it. In fact, `OpenRgn` calls `HidePen`, so no drawing occurs on the screen while the region is open (unless you called `ShowPen` just after `OpenRgn`, or you called `ShowPen` previously without balancing it by a call to `HidePen`). Since the pen hangs below and to the right of the pen location, drawing lines with even the smallest pen will change bits that lie outside the region you define.

The outline of a region is mathematically defined and infinitely thin, and separates the bitmap into two groups of bits: those within the region and those outside it. A region should consist of one or more closed loops. Each framed shape itself constitutes a loop. Any lines drawn with `Line` or `LineTo` should connect with each other or with a framed shape. Even though the on-screen presentation of a region is clipped, the definition of a region is not; you can define a region anywhere on the coordinate plane with complete disregard for the location of various `grafPort` entities on that plane.

When a region is open, the current `grafPort`'s `rgnSave` field contains a handle to information related to the region definition. If you want to temporarily disable the collection of lines and shapes, you can save the current value of this field, set the field to nil, and later restore the saved value to resume the region definition.

Warning: Do not call `OpenRgn` while another region is already open. All open regions but the most recent will behave strangely.

procedure `CloseRgn` (*dstRgn* : *RgnHandle*);

`CloseRgn` stops the collection of lines and framed shapes, organizes them into a region definition, and saves the resulting region into the region indicated by *dstRgn*. You should perform one and only one `CloseRgn` for every `OpenRgn`. `CloseRgn` calls `ShowPen`, balancing the `HidePen` call made by `OpenRgn`.

Here's an example of how to create and open a region, define a barbell shape, close the region, and draw it:

```

barbell := NewRgn;           {make a new region}
OpenRgn;                     {begin collecting stuff}
  SetRect(tempRect,20,20,30,50); {form the left weight}
  FrameOval(tempRect);
  SetRect(tempRect,30,30,80,40); {form the bar}
  FrameRect(tempRect);
  SetRect(tempRect,80,20,90,50); {form the right weight}
  FrameOval(tempRect);
CloseRgn(barbell);           {all done; save in barbell}
FillRgn(barbell,black);      {draw it on the screen}
DisposeRgn(barbell);         {don't need you anymore...}

```

procedure OffsetRgn(rgn : RgnHandle; dh, dv : integer);

OffsetRgn moves the region on the coordinate plane, a distance of *dh* horizontally and *dv* vertically. This does not affect the screen unless you subsequently call a routine to draw the region. If *dh* and *dv* are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The region retains its size and shape.

Note: OffsetRgn is an especially efficient operation, because most of the data defining a region is stored relative to *rgnBBox* and so isn't actually changed by OffsetRgn.

procedure MapRgn(rgn : RgnHandle; srcRect, dstRect : Rect);

Given a region within *srcRect*, MapRgn maps it to a similarly located region within *dstRect* by calling MapPt to map all the points in the region.

procedure InsetRgn(rgn : RgnHandle; dh, dv : integer);

InsetRgn shrinks or expands the region. All points on the region boundary are moved inwards a distance of *dv* vertically and *dh* horizontally; if *dh* or *dv* is negative, the points are moved outwards in that direction. InsetRgn leaves the region "centered" at the same position, but moves the outline in (for positive values of *dh* and *dv*) or out (for negative values of *dh* and *dv*). InsetRgn of a rectangular region works just like InsetRect.

procedure SectRgn(srcRgnA, srcRgnB, dstRgn : RgnHandle);

SectRgn calculates the intersection of two regions and places the intersection in a third region. *This does not create the destination region:* you must use NewRgn to create *dstRgn* before you call SectRgn. The *dstRgn* can be one of the source regions, if desired.

If the regions do not intersect, or one of the regions is empty, the destination is set to the empty region (0,0,0,0).

procedure UnionRgn(srcRgnA, srcRgnB, dstRgn : RgnHandle);

UnionRgn calculates the union of two regions and places the union in a third region. *This does not create the destination region:* you must use NewRgn to create *dstRgn* before you call UnionRgn. The *dstRgn* can be one of the source regions, if desired.

If both regions are empty, the destination is set to the empty region (0,0,0,0).

procedure DiffRgn(srcRgnA, srcRgnB, dstRgn : RgnHandle);

DiffRgn subtracts *srcRgnB* from *srcRgnA* and places the difference in a third region. *This does not create the destination region:* you must use NewRgn to create *dstRgn* before you call DiffRgn. The *dstRgn* can be one of the source regions, if desired.

If the first source region is empty, the destination is set to the empty region (0,0,0,0).

procedure *XorRgn*(*srcRgnA*, *srcRgnB*, *dstRgn* : *RgnHandle*);

XorRgn calculates the difference between the union and the intersection of two regions and places the result in a third region. *This does not create the destination region*: you must use *NewRgn* to create *dstRgn* before you call *XorRgn*. The *dstRgn* can be one of the source regions, if desired.

If the regions are coincident, the destination is set to the empty region (0,0,0,0).

function *PtInRgn*(*pt* : *Point*; *rgn* : *RgnHandle*) : *boolean*;

PtInRgn checks whether the pixel below and to the right of the given coordinate point is within the specified region, and returns *true* if so or *false* if not.

function *RectInRgn*(*r* : *Rect*; *rgn* : *RgnHandle*) : *boolean*;

RectInRgn checks whether the given rectangle intersects the specified region, and returns *true* if the intersection encloses at least one bit or *false* if not.

function *EqualRgn*(*rgnA*, *rgnB* : *rgnHandle*) : *boolean*;

EqualRgn compares the two regions and returns *true* if they are equal or *false* if not. The two regions must have identical sizes, shapes, and locations to be considered equal. Any two empty regions are always equal.

function *EmptyRgn*(*rgn* : *RgnHandle*) : *boolean*;

EmptyRgn returns *true* if the region is an empty region or *false* if not. Some of the circumstances in which an empty region can be created are: a *NewRgn* call; a *CopyRgn* of an empty region; a *SetRectRgn* or *RectRgn* with an empty rectangle as an argument; *CloseRgn* without a previous *OpenRgn* or with no drawing after an *OpenRgn*; *OffsetRgn* of an empty region; *InsetRgn* with an empty region or too large an inset; *SectRgn* of nonintersecting regions; *UnionRgn* of two empty regions; and *DiffRgn* or *XorRgn* of two identical or nonintersecting regions.

C.8.13 Graphic Operations on Regions

These routines all depend on the coordinate system of the current *grafPort*. If a region is drawn in a different *grafPort* than the one in which it was defined, it may not appear in the proper position inside the port.

procedure *FrameRgn*(*rgn* : *RgnHandle*);

FrameRgn draws a hollow outline just inside the specified region, using the current *grafPort*'s pen pattern, mode, and size. The outline is as wide as the pen width and as tall as the pen height;

under no circumstances will the frame go outside the region boundary. The pen location is not changed by this procedure.

If a region is open and being formed, the outside outline of the region being framed is mathematically added to that region's boundary.

procedure *PaintRgn*(*rgn* : *RgnHandle*);

PaintRgn paints the specified region with the current *grafPort*'s pen pattern and pen mode. The region on the bitmap is filled with the *pnPat*, according to the pattern transfer mode specified by *pnMode*. The pen location is not changed by this procedure.

procedure *EraseRgn*(*rgn* : *RgnHandle*);

EraseRgn paints the specified region with the current *grafPort*'s background pattern *bkPat* (in *patCopy* mode). The *grafPort*'s *pnPat* and *pnMode* are ignored; the pen location is not changed.

procedure *InvertRgn*(*rgn* : *RgnHandle*);

InvertRgn inverts the pixels enclosed by the specified region: every white pixel becomes black and every black pixel becomes white. The *grafPort*'s *pnPat*, *pnMode*, and *bkPat* are all ignored; the pen location is not changed.

procedure *FillRgn*(*rgn* : *RgnHandle*; *pat* : *Pattern*);

FillRgn fills the specified region with the given pattern (in *patCopy* mode). The *grafPort*'s *pnPat*, *pnMode*, and *bkPat* are all ignored; the pen location is not changed.

C.8.14 Bit Transfer Operations

procedure *ScrollRect*(*r* : *Rect*; *dh*, *dv* : *integer*;
 updateRgn : *RgnHandle*);

ScrollRect shifts ("scrolls") those bits inside the intersection of the specified rectangle, *visRgn*, *clipRgn*, *portRect*, and *portBits.bounds*. The bits are shifted a distance of *dh* horizontally and *dv* vertically. The positive directions are to the right and down. No other bits are affected. Bits that are shifted out of the scroll area are lost; they are neither placed outside the area nor saved. The *grafPort*'s background pattern *bkPat* fills the space created by the scroll. In addition, *updateRgn* is changed to the area filled with *bkPat* (see Figure C-23).

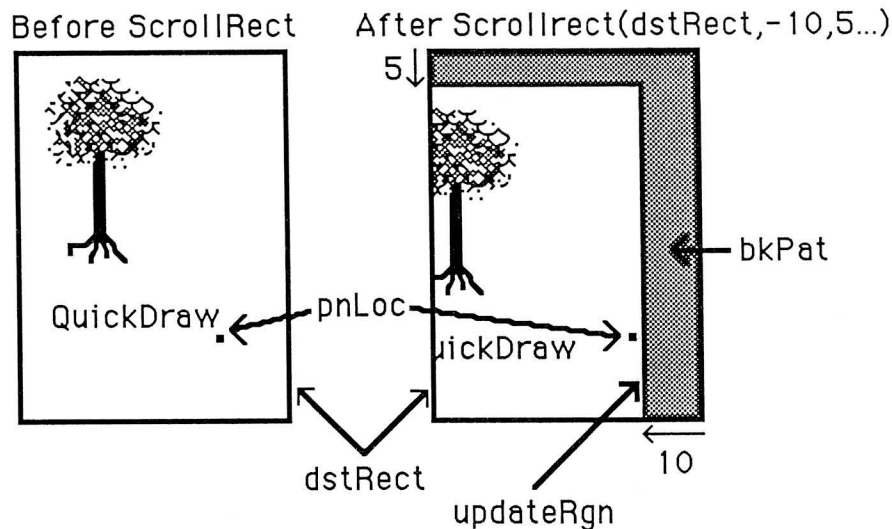


Figure C-23
Scrolling

Figure C-23 shows that the pen location after a ScrollRect is in a different position relative to what was scrolled in the rectangle. The entire scrolled item has been moved to different coordinates. To restore it to its coordinates before the ScrollRect, you can use the SetOrigin procedure. For example, suppose the *dstRect* here is the *portRect* of the grafPort and its top left corner is at (95,120). SetOrigin(105,115) will offset the coordinate system to compensate for the scroll. Since the *clipRgn* and pen location are not offset, they move down and to the left.

```
procedure CopyBits( srcBits, dstBits : BitMap;
                    srcRect, dstRect : Rect;
                    mode : integer; maskRgn : RgnHandle );
```

CopyBits transfers a bit image between any two bitmaps and clips the result to the area specified by the *maskRgn* parameter. The transfer may be performed in any of the eight source transfer modes. The result is always clipped to the *maskRgn* and the boundary rectangle of the destination bitmap; if the destination bitmap is the current grafPort's *portBits*, it is also clipped to the intersection of the grafPort's *clipRgn* and *visRgn*. If you do not want to clip to a *maskRgn*, just pass *nil* for the *maskRgn* parameter.

The *dstRect* and *maskRgn* coordinates are in terms of the *dstBits.bounds* coordinate system, and the *srcRect* coordinates are in terms of the *srcBits.bounds* coordinates.

The bits enclosed by the source rectangle are transferred into the destination rectangle according to the rules of the chosen mode.

The source transfer modes are as follows:

<i>srcCopy</i>	<i>srcXor</i>	<i>notSrcCopy</i>	<i>notSrcXor</i>
<i>srcOr</i>	<i>srcBic</i>	<i>notSrcOr</i>	<i>notSrcBic</i>

The source rectangle is completely aligned with the destination rectangle; if the rectangles are of different sizes, the bit image is expanded or shrunk as necessary to fit the destination rectangle. For example, if the bit image is a circle in a square source rectangle, and the destination rectangle is not square, the bit image appears as an oval in the destination (see Figure C-24).

C.8.15 Pictures

function *OpenPicture*(*picFrame* : *Rect*) : *PicHandle*;

OpenPicture returns a handle to a new picture which has the given rectangle as its picture frame, and tells QuickDraw to start saving as the picture definition all calls to drawing routines and all picture comments (if any).

OpenPicture calls *HidePen*, so no drawing occurs on the screen while the picture is open (unless you call *ShowPen* just after *OpenPicture*, or you called *ShowPen* previously without balancing it by a call to *HidePen*).

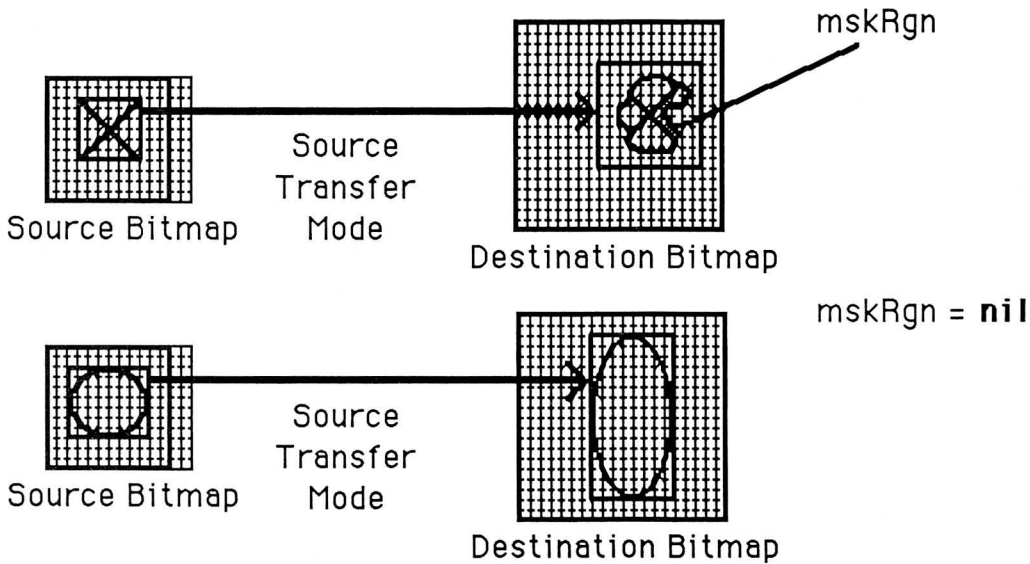


Figure C-24
Operation of CopyBits

When a picture is open, the current grafPort's *picSave* field contains a handle to information related to the picture definition. If you want to temporarily disable the collection of routine calls and picture comments, you can save the current value of this field, set the field to *nil*, and later restore the saved value to resume the picture definition.

Warning: Do not call *OpenPicture* while another picture is already open.

procedure *ClosePicture*;

ClosePicture tells QuickDraw to stop saving routine calls and picture comments as the definition of the currently open picture. You should perform one and only one *ClosePicture* for every *OpenPicture*. *ClosePicture* calls *ShowPen*, balancing the *HidePen* call made by *OpenPicture*.

procedure *PicComment*(*kind*, *dataSize* : integer;
 dataHandle : Handle);

PicComment inserts the specified comment into the definition of the currently open picture. *Kind* identifies the type of comment. *DataHandle* is a handle to additional data if desired, and *dataSize* is the size of that data in bytes. If there is no additional data for the comment, *dataHandle* should be *nil* and *dataSize* should be 0. The application that processes the comment must include a procedure to do the processing and store a pointer to the procedure in the data structure pointed to by the *grafProcs* field of the *grafPort* (see Section C.9, Customizing QuickDraw Operations).

procedure *DrawPicture*(*myPicture* : *PicHandle*; *dstRect* : Rect);

DrawPicture draws the given picture to scale in *dstRect*, expanding or shrinking it as necessary to align the borders of the picture frame with *dstRect*. *DrawPicture* passes any picture comments to the procedure accessed indirectly through the *grafProcs* field of the *grafPort* (see *PicComment* above).

procedure *KillPicture*(*myPicture* : *PicHandle*);

KillPicture deallocates space for the picture whose handle is supplied, and returns the memory used by the picture to the free memory pool. Use this only when you are completely through with a picture.

C.8.16 Calculations with Polygons

function *OpenPoly* : *PolyHandle*;

OpenPoly returns a handle to a new polygon and tells QuickDraw to start saving the polygon definition as specified by calls to line-drawing routines. While a polygon is open, all calls to *Line* and *LineTo* affect the outline of the polygon. Only the line endpoints affect the polygon definition; the pen mode, pattern, and size do not affect it. In fact, *OpenPoly* calls *HidePen*, so no drawing occurs on the screen while the polygon is open (unless you call *ShowPen* just after *OpenPoly*, or you called *ShowPen* previously without balancing it by a call to *HidePen*).

A polygon should consist of a sequence of connected lines. Even though the on-screen presentation of a polygon is clipped, the definition of a polygon is not; you can define a polygon anywhere on the coordinate plane with complete disregard for the location of various *grafPort* entities on that plane.

When a polygon is open, the current grafPort's *polySave* field contains a handle to information related to the polygon definition. If you want to temporarily disable the polygon definition, you can save the current value of this field, set the field to *nil*, and later restore the saved value to resume the polygon definition.

Warning: Do not call *OpenPoly* while another polygon is already open.

procedure *ClosePoly*;

ClosePoly tells *QuickDraw* to stop saving the definition of the currently open polygon and computes the *polyBBox* rectangle. You should perform one and only one *ClosePoly* for every *OpenPoly*. *ClosePoly* calls *ShowPen*, balancing the *HidePen* call made by *OpenPoly*.

Here's an example of how to open a polygon, define it as a triangle, close it, and draw it:

```
triPoly := OpenPoly;           { save handle, begin collecting }
  MoveTo(300,100);             { move to first point and }
  LineTo(400,200);             {      form      }
  LineTo(200,200);             {      the      }
  LineTo(300,100);             {      triangle   }
ClosePoly;                     { stop collecting stuff  }
FillPoly(triPoly,gray);        { draw it on the screen  }
KillPoly(triPoly);             { we're all done   }
```

procedure *KillPoly*(*poly* : *PolyHandle*);

KillPoly deallocates space for the polygon whose handle is supplied, and returns the memory used by the polygon to the free memory pool. Use this only after you are completely through with a polygon.

procedure *OffsetPoly*(*poly* : *PolyHandle*; *dh*, *dv* : *integer*);

OffsetPoly moves the specified polygon on the coordinate plane, a distance of *dh* horizontally and *dv* vertically. This does not affect the screen unless you subsequently call a routine to draw the polygon. If *dh* and *dv* are positive, the movement is to the right and down; if either is negative, the corresponding movement is in the opposite direction. The polygon retains its shape and size.

Note: *OffsetPoly* is an especially efficient operation, because the data defining a polygon is stored relative to *polyStart* and so isn't actually changed by *OffsetPoly*.

C.8.17 Graphic Operations on Polygons

procedure *FramePoly*(*poly* : *PolyHandle*);

FramePoly plays back the line-drawing routine calls that define the given polygon, using the current *grafPort*'s pen pattern, mode, and size. The pen will hang below and to the right of each point on the boundary of the polygon; thus, the polygon drawn will extend beyond the right and bottom edges of *poly*^{^^}.*polyBBox* by the pen width and pen height, respectively. All other graphic operations occur strictly within the boundary of the polygon, as for other shapes. You can see this difference in Figure C-25, where each of the polygons is shown with its *polyBBox*.

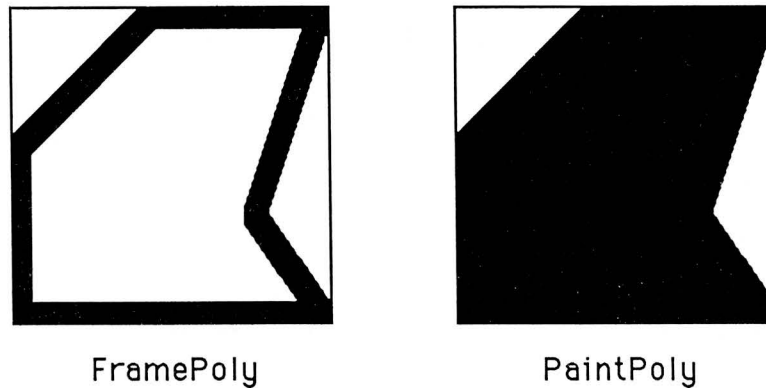


Figure C-25
Drawing Polygons

If a polygon is open and being formed, *FramePoly* affects the outline of the polygon just as if the line-drawing routines themselves had been called. If a region is open and being formed, the outside outline of the polygon being framed is mathematically added to the region's boundary.

procedure *PaintPoly*(*poly* : *PolyHandle*);

PaintPoly paints the specified polygon with the current *grafPort*'s pen pattern and pen mode. The polygon on the bitmap is filled with the *pnPat*, according to the pattern transfer mode specified by *pnMode*. The pen location is not changed by this procedure.

procedure *ErasePoly*(*poly* : *PolyHandle*);

ErasePoly paints the specified polygon with the current *grafPort*'s background pattern *bkPat* (in *patCopy* mode). The *pnPat* and *pnMode* are ignored; the pen location is not changed.

procedure *InvertPoly*(*poly* : *PolyHandle*);

InvertPoly inverts the pixels enclosed by the specified polygon: every white pixel becomes black and every black pixel becomes white. The *grafPort*'s *pnPat*, *pnMode*, and *bkPat* are all ignored; the pen location is not changed.

procedure *FillPoly*(*poly* : *PolyHandle*; *pat* : *Pattern*);

FillPoly fills the specified polygon with the given pattern (in *patCopy* mode). The *grafPort*'s *pnPat*, *pnMode*, and *bkPat* are all ignored; the pen location is not changed.

C.8.18 Miscellaneous Utilities

function *Random* : *integer*;

Random returns an integer, uniformly distributed pseudo-random, in the range from -32768 through 32767. The value returned depends on the global variable *randSeed*, which *InitGraf* initializes to 1; you can start the sequence over again from where it began by resetting *randSeed* to 1.

function *GetPixel*(*h*, *v* : *integer*) : *boolean*;

GetPixel looks at the pixel associated with the given coordinate point and returns *true* if it is black or *false* if it is white. The selected pixel is immediately below and to the right of the point whose coordinates are given in *h* and *v*, in the local coordinates of the current *grafPort*. There is no guarantee that the specified pixel actually belongs to the port, however; it may have been drawn by a port overlapping the current one. To see if the point indeed belongs to the current port, call *PtInRgn*(*pt*, *thePort*^.*visRgn*).

procedure *StuffHex*(*thingPtr* : *Ptr*; *s* : *Str255*);

StuffHex pokes bits (expressed as a string of hexadecimal digits) into any data structure. This is a good way to create cursors, patterns, or bit images to be "stamped" onto the screen with *CopyBits*. For example,

```
StuffHex(@stripes, '0102040810204080')
```

places a striped pattern into the pattern variable *stripes*.

Warning: There is no range checking on the size of the destination variable. It's easy to overrun the variable and destroy something if you don't know what you're doing.

C.9 Customizing QuickDraw Operations

For each shape that QuickDraw knows how to draw, there are procedures that perform these basic graphic operations on the shape: frame, paint, erase, invert, and fill. Those procedures in turn call a low-level drawing routine for the shape. For example, the `FrameOval`, `PaintOval`, `EraseOval`, `InvertOval`, and `FillOval` procedures all call a low-level routine that draws the oval. For each type of object QuickDraw can draw, including text and lines, there is a pointer to such a routine. By changing these pointers, you can install your own routines, and either completely override the standard ones or call them after your routines have modified parameters as necessary.

Other low-level routines that you can install in this way are:

- The procedure that does bit transfer and is called by `CopyBits`.
- The function that measures the width of text and is called by `CharWidth`, `StringWidth`, and `TextWidth`.
- The procedure that processes picture comments and is called by `DrawPicture`. The standard such procedure ignores picture comments.
- The procedure that saves drawing commands as the definition of a picture, and the one that retrieves them. This enables the application to draw on remote devices, print to the disk, get picture input from the disk, and support large pictures.

The `grafProcs` field of a `grafPort` determines which low-level routines are called; if it contains ***nil***, the standard routines are called, so that all operations in that `grafPort` are done in the standard ways described in this appendix. You can set the `grafProcs` field to point to a record of pointers to routines. The data type of `grafProcs` is `QDProcsPtr`:

```
type ProcsPtr = QDProcs;
QDProcs = record
    textProc: Ptr;      {text drawing}
    lineProc: Ptr;      {line drawing}
    rectProc: Ptr;      {rectangle drawing}
    rRectProc: Ptr;     {roundRect drawing}
    ovalProc: Ptr;      {oval drawing}
    arcProc: Ptr;       {arc/wedge drawing}
    polyProc: Ptr;      {polygon drawing}
    rgnProc: Ptr;       {region drawing}
    bitsProc: Ptr;      {bit transfer}
    commentProc: Ptr;   {picture comment processing}
    txMeasProc: Ptr;    {text width measurement}
    getPicProc: Ptr;    {picture retrieval}
    putPicProc: Ptr     {picture saving}
end;
```

```
procedure SetStdProcs( var procs : QDProcs );
```

`SetStdProcs` is provided to assist you in setting up a `QDProcs` record. It sets all the fields of the given `QDProcs` to point to the standard low-level routines. You can then change the ones you

wish to point to your own routines. For example, if your procedure that processes picture comments is named *MyComments*, you will store *@MyComments* in the *commentProc* field of the *QDProcs* record.

The routines you install must of course have the same calling sequences as the standard routines, which are described below. The standard drawing routines tell which graphic operation to perform from a parameter of type *GrafVerb*.

```
type GrafVerb = (frame, paint, erase, invert, fill);
```

When the *grafVerb* is *fill*, the pattern to use when filling is passed in the *fillPat* field of the *grafPort*.

```
procedure StdText( byteCount      : integer; textBuf : Ptr;  
                   numer, denom : Point );
```

StdText is the standard low-level routine for drawing text. It draws text from the arbitrary structure in memory specified by *textBuf*, starting from the first byte and continuing for *byteCount* bytes. *Numer* and *denom* specify the scaling, if any: *numer.v* over *denom.v* gives the vertical scaling, and *numer.h* over *denom.h* gives the horizontal scaling.

```
procedure StdLine( newPt : Point );
```

StdLine is the standard low-level routine for drawing a line. It draws a line from the current pen location to the location specified (in local coordinates) by *newPt*.

```
procedure StdRect( verb : GrafVerb; r : Rect );
```

StdRect is the standard low-level routine for drawing a rectangle. It draws the given rectangle according to the specified *grafVerb*.

```
procedure StdRRect( verb : GrafVerb; r : Rect;  
                   ovalwidth, ovalHeight : integer );
```

StdRRect is the standard low-level routine for drawing a rounded-corner rectangle. It draws the given rounded-corner rectangle according to the specified *grafVerb*. *OvalWidth* and *ovalHeight* specify the diameters of curvature for the corners.

```
procedure StdOval( verb : GrafVerb; r : Rect );
```

StdOval is the standard low-level routine for drawing an oval. It draws an oval inside the given rectangle according to the specified *grafVerb*.

```
procedure StdArc( verb : GrafVerb; r : Rect;
                  startAngle, arcAngle : integer );
```

StdArc is the standard low-level routine for drawing an arc or a wedge. It draws an arc or wedge of the oval that fits inside the given rectangle. The *grafVerb* specifies the graphic operation; if it's the frame operation, an arc is drawn; otherwise, a wedge is drawn.

```
procedure StdPoly( verb : GrafVerb; poly : PolyHandle );
```

StdPoly is the standard low-level routine for drawing a polygon. It draws the given polygon according to the specified *grafVerb*.

```
procedure StdRgn( verb : GrafVerb; rgn : RgnHandle );
```

StdRgn is the standard low-level routine for drawing a region. It draws the given region according to the specified *grafVerb*.

```
procedure StdBits( var srcBits : BitMap;
                   var srcRect, dstRect : Rect; mode : integer;
                   maskRgn : RgnHandle );
```

StdBits is the standard low-level routine for doing bit transfer. It transfers a bit image between the given bitmap and *thePort^.portBits*, just as if CopyBits were called with the same parameters and with a destination bitmap equal to *thePort^.portBits*.

```
procedure StdComment( kind, dataSize : integer;
                      dataHandle : Handle );
```

StdComment is the standard low-level routine for processing a picture comment. *Kind* identifies the type of comment. *DataHandle* is a handle to additional data, and *dataSize* is the size of that data in bytes. If there is no additional data for the command, *dataHandle* will be *nil* and *dataSize* will be 0. StdComment simply ignores the comment.

```
function StdTxMeas( byteCount : integer;
                    textBuf : Ptr;
                    var numer, denom : Point;
                    var info : FontInfo ) : integer;
```

StdTxMeas is the standard low-level routine for measuring text width. It returns the width of the text stored in the arbitrary structure in memory specified by *textBuf*, starting with the first byte and continuing for *byteCount* bytes. *Numer* and *denom* specify the scaling as in the StdText procedure; note that StdTxMeas may change them.

procedure *StdGetPic*(*dataPtr* : *Ptr*; *byteCount* : *integer*);

StdGetPic is the standard low-level routine for retrieving information from the definition of a picture. It retrieves the next *byteCount* bytes from the definition of the currently open picture and stores them in the data structure pointed to by *dataPtr*.

procedure *StdPutPic*(*dataPtr* : *Ptr*; *byteCount* : *integer*);

StdPutPic is the standard low-level routine for saving information as the definition of a picture. It saves as the definition of the currently open picture the drawing commands stored in the data structure pointed to by *dataPtr*, starting with the first byte and continuing for the next *byteCount* bytes.

Appendix D

The Standard Apple Numeric Environment (SANE)

This appendix contains a reproduction of an Apple numerics manual entitled *The Standard Apple Numeric Environment*.

The numerics manual is a language-independent description of the Standard Apple Numeric Environment (SANE). Lightspeed Pascal supports parts of this environment through its built-in arithmetic, and supports other parts through the SANE library. Thus some of the features described in the numerics manual (for example, SANE data formats, extended-based expression evaluation, and the mathematical functions which are part of standard Pascal) are an integral part of Lightspeed Pascal; access to these features is described in the Language Reference. Other features described in the numerics manual (for example, exception-handling facilities and some elementary functions) are provided in the SANE library.

To use the SANE library, simply include the name SANE in a uses-clause (see Section 8.3 of the Language Reference), e.g.

```
uses SANE;
```

You must also add the library file *SANELib* and the interface file *SANE* to your project --see Chapter 8 of the User's Guide. A list of the interface to the SANE library can be found in Appendix E.

Be aware that the SANE data types *single* and *comp* correspond to the Lightspeed Pascal real-types *real* and *computational* respectively. Lightspeed Pascal will accept all of these names.

All handling of real-types in Lightspeed Pascal is done according to the Standard Apple Numeric Environment, regardless of whether or not SANE appears in a uses-clause. However, by including SANE in a uses-clause, the additional declarations in the SANE library become available.

Appendix D

The Standard Apple Numeric Environment (SANE)

D.1	Introduction	D-5
D.2	Data Types	D-5
D.2.1	Choosing a Data Type	D-6
D.2.2	Values Represented	D-6
D.2.3	Range and Precision of SANE Types	D-6
D.2.4	Formats	D-8
D.3	Arithmetic Operations	D-9
D.3.1	Remainder	D-10
D.3.2	Round to Integral Value	D-11
D.4	Conversions	D-11
D.4.1	Conversions Between Extended and Single or Double	D-11
D.4.2	Conversions to Comp and Other Integral Formats	D-11
D.4.3	Conversions Between Binary and Decimal	D-12
D.4.4	Conversions Between Decimal Formats	D-15
D.5	Expression Evaluation	D-15
D.5.	Using Extended Temporaries	D-16
D.5.2	Extended-Precision Expression Evaluation	D-16
D.5.3	Extended-Precision Expression Evaluation and the IEEE Standard	D-16
D.6	Comparisons	D-17
D.7	Infinities, NaNs, and Denormalized Numbers	D-18
D.7.1	Infinities	D-18
D.7.2	NaNs	D-18
D.7.3	Denormalized Numbers	D-19

D.7.4	Inquiries: Class and Sign	D-20
D.8	Environmental Control	D-20
D.8.1	Rounding Direction	D-21
D.8.2	Rounding Precision	D-21
D.8.3	Exception Flags and Halts	D-21
D.8.4	Managing Environmental Settings	D-23
D.9	Auxiliary Procedures	D-25
D.9.1	Sign Manipulation	D-25
D.9.2	Next-After Functions	D-25
D.9.3	Binary Scale and Log Functions	D-26
D.10	Elementary Functions	D-26
D.10.1	Logarithm Functions	D-26
D.10.2	Exponential Functions	D-27
D.10.3	Financial Functions	D-28
D.10.4	Trigonometric Functions	D-29
D.10.5	Random Number Generator	D-30
D.11	Annotated Bibliography	D-31
D.12	Glossary	D-33

The Standard Apple Numeric Environment (SANE)

D.1 Introduction

This appendix describes the Standard Apple Numeric Environment (SANE). Apple supports SANE on several current products and plans to support SANE on future products. SANE gives you access to numeric facilities unavailable on almost any computer of the early 1980's--from microcomputers to extremely fast, extremely expensive supercomputers. The core features of SANE are not exclusive to Apple; rather they are taken from Draft 10.0 of Standard 754 for Binary Floating-Point Arithmetic [11] as proposed to the Institute of Electrical and Electronics Engineers (IEEE). Thus SANE is one of the first widely available products with the arithmetic capabilities destined to be found on the computers of the mid-1980's and beyond.

The IEEE Standard specifies standardized data types, arithmetic, and conversions, along with tools for handling limitations and exceptions, that are sufficient for numeric applications. SANE supports all requirements of the IEEE Standard. SANE goes beyond the specifications of the Standard by including a data type designed for accounting applications and by including several high-quality library functions for financial and scientific calculations. IEEE arithmetic was specifically designed to provide advanced features for numerical analysts without imposing extra burden on casual users. (This is an admirable but rarely attainable goal: text editors and word processors, for example, typically suffer increased complexity with added features, meaning more hurdles for the novice to clear before completing even the simplest tasks.) The independence of elementary and advanced features of the IEEE arithmetic was carried over to SANE.

D.2 Data Types

SANE provides three *application* data types (single, double, and comp) and the *arithmetic* type (extended). Single, double, and extended store floating-point values and comp stores integral values.

The *extended* type is called the arithmetic type because, to make expression evaluation simpler and more accurate, SANE performs all arithmetic operations in extended precision and delivers arithmetic results to the extended type. *Single*, *double*, and *comp* can be thought of as space-saving storage types for the extended-precision arithmetic. (In this manual, we shall use the term *extended precision* to denote both the extended precision and the extended range of the extended type.)

All values representable in single, double, and comp (as well as 16-bit and 32-bit integers) can be represented exactly in extended. Thus values can be moved from any of these types to the extended type and back without any loss of information.

D.2.1 Choosing a Data Type

Typically, picking a data type requires that you determine the trade-offs between

- fixed- or floating-point form
- precision
- range
- memory usage
- speed

The precision, range, and memory usage for each SANE data type are shown in Table 2-1. Effects of the data types on performance (speed) vary among the implementations of SANE. (See Section D.4 for information on conversion problems relating to precision.)

Most accounting applications require a counting type that counts things (pennies, dollars, widgets) exactly. Accounting applications can be implemented by representing money values as integral numbers of cents or mils, which can be stored exactly in the storage format of the *comp* (for computational) type. The sum, difference, or product of any two comp values is exact if the magnitude of the result does not exceed $2^{63} - 1$ (that is, 9,223,372,036,854,775,807). This number is larger than the U.S. national debt expressed in Argentine pesos. In addition, comp values (such as the results of accounting computations) can be mixed with extended values in floating-point computations (such as compound interest).

Arithmetic with comp-type variables, like all SANE arithmetic, is done internally using extended-precision arithmetic. There is no loss of precision, as conversion from comp to extended is always exact. Space can be saved by storing numbers in the comp type, which is 20 percent shorter than extended. Non-accounting applications will normally be better served by the floating-point data formats.

D.2.2 Values Represented

The floating-point storage formats (single, double, and extended) provide binary encodings of a *sign* (+ or -), an *exponent*, and a *significand*. A represented number has the value

$$\pm \text{significand} * 2^{\text{exponent}}$$

where the significand has a single bit to the left of the binary point (that is, $0 \leq \text{significand} < 2$).

D.2.3 Range and Precision of SANE Types

This table describes the range and precision of the numeric data types supported by SANE. Decimal ranges are expressed as chopped two-digit decimal representations of the exact binary values.

Usually numbers are stored in a *normalized* form, to afford maximum precision for a given significant width. Maximum precision is achieved if the high order bit in the significant is 1 (that is, $1 \leq \text{significant} < 2$).

Type class				
Type identifier	Single	Double	Comp	Extended
Size (bytes:bits)	4:32	8:64	8:64	10:80
Binary exponent range				
Minimum	-126	-1022	----	-16383
Maximum	127	1023	----	16384
Significand precision				
Bits	24	53	63	64
Decimal digits	7-8	15-16	18-19	19-20
Decimal range (approximate)				
Min negative	-3.4E+38	-1.7E+308	$\approx -9.2E+18$	-1.1E+4932
Max neg norm	-1.2E-38	-2.3E-308		-1.7E-4932
Max neg denorm*	-1.5E-45	-5.0E-324		-1.9E-4951
Min pos denorm*	1.5E-45	5.0E-324		1.9E-4951
Min pos norm	1.2E-38	2.3E-308		1.7E-4932
Max positive	3.4E+38	1.7E+308	$\approx 9.2E+18$	1.1E+4932
Infinites*	Yes	Yes	No	Yes
NaNs	Yes	Yes	Yes	Yes

* *Denorms* (denormalized numbers), *NaNs* (Not-a-Number), and *infinities* are defined in Section 7.

Table D-1
SANE Types

Example

In Single, the largest representable number has

[illegible]

the smallest representable positive normalized number has

[illegible]

and the smallest representable positive denormalized number (see Section D.7) has

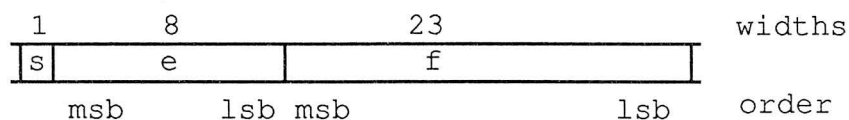
[illegible]

D.2.4 Formats

This section shows the formats of the four SANE numeric data types. These are pictorial representations and may not reflect the actual byte order in any particular implementation.

Single

A 32-bit single format number is divided into three fields as shown below.



The value v of the number is determined by these fields as follows:

```

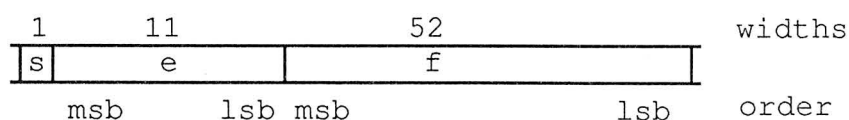
if 0 < e < 255,      then v =  $(-1)^s * 2^{(e-127)} * (1.f)$ ;
if e = 0 and f ≠ 0, then v =  $(-1)^s * 2^{(-126)} * (0.f)$ ;
if e = 0 and f = 0, then v =  $(-1)^s * 0$ ;
if e = 255 and f = 0, then v =  $(-1)^s * \infty$ ;
if e = 255 and f ≠ 0, then v is a NaN.

```

See Section D.7 for information on the contents of the f field for NaNs.

Double

A 64-bit double format number is divided into three fields as shown below.



The value v of the number is determined by these fields as follows:

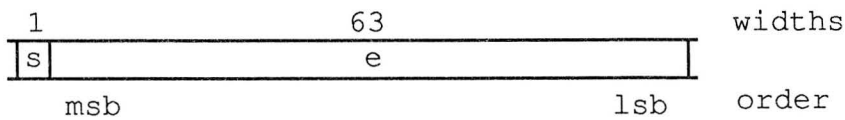
```

if  $0 < e < 2047$ ,          then  $v = (-1)^s * 2^{(e-1023)} * (1.f)$ ;
if  $e = 0$  and  $f \neq 0$ , then  $v = (-1)^s * 2^{(-1022)} * (0.f)$ ;
if  $e = 0$  and  $f = 0$ , then  $v = (-1)^s * 0$ ;
if  $e = 2047$  and  $f = 0$ , then  $v = (-1)^s * \infty$ ;
if  $e = 2047$  and  $f \neq 0$ , then  $v$  is a NaN.

```

Comp

A 64-bit comp format number is divided into two fields as shown below.



The value v of the number is determined by these fields as follows:

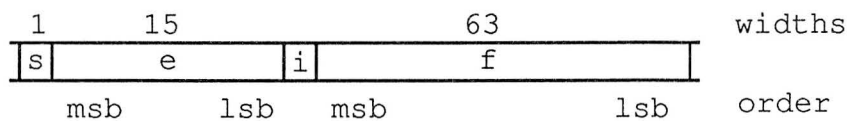
```

if  $s = 1$  and  $d = 0$ , then  $v$  is the unique comp NaN;
otherwise,  $v$  is the two's-complement value of the
64-bit representation.

```

Extended

An 80-bit extended format number is divided into four fields as shown below.



The value v of the number is determined by these fields as follows:

```

if  $0 \leq e < 32767$ ,          then  $v = (-1)^s * 2^{(e-16383)} * (i.f)$ ;
if  $e = 32767$  and  $f = 0$ , then  $v = (-1)^s * \infty$ , regardless of  $i$ ;
if  $e = 32767$  and  $f \neq 0$ , then  $v$  is a NaN, regardless of  $i$ .

```

D.3 Arithmetic Operations

SANE provides the basic arithmetic operations:

- add
- subtract
- multiply
- divide

- square root
- remainder
- round to integral value

for the SANE data types. (See Sections D.9 and D.10 for auxiliary operations and higher-level functions supported by SANE.)

All the basic arithmetic operations produce the best possible result: the mathematically exact result coerced to the precision and range of the extended type. The coercions honor the user-selectable rounding direction and handle all exceptions according to the requirements of the IEEE Standard (see Section D.8).

D.3.1 Remainder

Generally, remainder (and mod) functions are defined by the expression

$$x \text{ rem } y = x - y * n$$

where n is some integral approximation to the quotient x/y . This expression can be found even in the conventional integer-division algorithm:

$$\begin{array}{rcl}
 & & n \quad (\text{integral quotient approximation}) \\
 (\text{divisor}) & y) \overline{x} & (\text{dividend}) \\
 & \quad y * n & \\
 & \quad \text{-----} & \\
 & \quad x - y * n & (\text{remainder})
 \end{array}$$

SANE supports the remainder function specified in the IEEE Standard:

When $y \neq 0$, the remainder $r = x \text{ rem } y$ is defined regardless of the rounding direction by the mathematical relation $r = x - y * n$, where n is the integral value nearest the exact value x/y ; whenever $|n - x/y| = 1/2$, n is even. The remainder is always exact. If $r = 0$, its sign is that of x .

Example 1: $5 \text{ rem } 3$

Here $x = 5$ and $y = 3$. Since $1 < 5/3 < 2$ and since $5/3 = 1.66666\dots$ is closer to 2 than to 1, n is taken to be 2, so

$$5 \text{ rem } 3 = r = 5 - 3 * 2 = -1$$

Example 2: $7.0 \text{ rem } 0.4$

Since $17 < 7.0/0.4 < 18$ and since $7.0/0.4 = 17.5$ is equally close to both 17 and 18, n is taken to be the even quotient, 18. Hence,

$$7.0 \text{ rem } 0.4 = r = 7.0 - 0.4 * 18 = -0.2$$

The IEEE remainder function differs from other commonly used remainder and mod functions. It returns a remainder of the smallest possible magnitude, and it always returns an exact remainder.

All the other remainder functions can be constructed from the IEEE remainder.

D.3.2 Round to Integral Value

An input argument is rounded according to the current rounding direction to an integral value and delivered to the extended format. For example, 12345678.875 rounds to 12345678.0 or 12345679.0. (The rounding direction, which can be set by the user, is explained fully in Section D.8.)

Note that, in each floating-point format, all values of sufficiently great magnitude are integral. For example, in single, numbers whose magnitudes are at least 2^{23} are integral.

D.4 Conversions

SANE provides conversions between the extended type and each of the other SANE types (single, double, and comp). A particular SANE implementation will provide conversions between extended and other numeric types supported in its particular larger environment. For example, a Pascal implementation will have conversions between extended and the Pascal integer type.

SANE implementations also provide either conversions between decimal strings and SANE types, or conversions between a decimal record type and SANE types, or both. Conversions between decimal records and decimal strings may be included too.

D.4.1 Conversions Between Extended and Single or Double

A conversion to extended is always exact. A conversion from extended to single or double moves a value to a storage type with less range and precision, and sets the overflow, underflow, and inexact exception flags as appropriate. (See Section D.8 for a discussion of exception flags.)

D.4.2 Conversions to Comp and Other Integral Formats

Conversions to integral formats are done by first rounding to an integral value (honoring the current rounding direction) and then, if possible, delivering this value to the destination format. If the source operand of a conversion from extended to comp is a NaN, an infinity, or out-of-range for the comp format, then the result is the comp NaN and for infinities and values out-of-range, the invalid exception is signaled. If the source operand of a conversion to a system-specific integer type is a NaN, infinity, or out-of-range for that format, then invalid is signaled (unless the type has an appropriate representation for the exceptional result). NaNs, infinities, and out-of-range values are stored in two's-complement integer formats as the extreme negative values (for example, in the 16-bit integer format, as -32768).

Note that IEEE rounding into integral formats differs from most common rounding functions on halfway cases. With the default rounding direction (to nearest), conversions to comp or to a system-specific integer type will round 0.5 to 0, 1.5 to 2, 2.5 to 2, and 3.5 to 4, rounding to even on halfway cases. (Rounding is discussed in detail in Section D.8.)

D.4.3 Conversions Between Binary and Decimal

The IEEE Standard for binary floating-point arithmetic specifies the set of numerical values representable within each floating-point format. It is important to recognize that binary storage formats can exactly represent the fractional part of decimal numbers in only a few cases; in all other cases, the representation will be approximate. For example, 0.5_{10} , or $1/2_{10}$, can be represented exactly as 0.1_2 . On the other hand, 0.1_{10} , or $1/10_{10}$, is a repeating fraction in binary: $0.00011001100\dots_2$. Its closest representation in Single is $0.000110011001100110011001101_2$, which is closer to 0.10000000149_{10} than to 0.10000000000_{10} .

As binary storage formats generally provide only close approximations to decimal values, it is important that conversions between the two types be as accurate as possible. Given a rounding direction, for every decimal value there is a best (correctly rounded) binary value for each binary format. Conversely, for any rounding direction, each binary value has a corresponding best decimal representation for a given decimal format. Ideally, binary-decimal conversions should obtain this best value to reduce accumulated errors. Conversion routines in SANE implementations meet or exceed the stringent error bounds specified by the IEEE Standard. This means that although in extreme cases the conversions do not deliver the correctly rounded result, the result delivered is very nearly as good as the correctly rounded result. (See the IEEE Standard [11] for a more detailed description of error bounds.)

Conversions from Decimal Strings to SANE Types

Routines may be provided to convert numeric decimal strings to the SANE data types. These routines are provided for the convenience of those who do not wish to write their own parsers and scanners. Examples of acceptable input are

123	123.4E-12	-123.	.456	3e9	-0
-INF	Inf	NAN(12)	-NaN()	nan	

The 12 in *NAN(12)* is a NaN code (see Section D.8).

The accepted syntax is formally defined, using Backus-Naur form, in Table D-2:

```
<decimal number>      ::= [{space | tab}] <left decimal>
<left decimal>        ::= [+|-] <unsigned decimal>
<unsigned decimal>    ::= <finite number> | <infinity> | <NAN>
<finite number>      ::= <significand> [<exponent>]
<significand>        ::= <integer> | <mixed>
<integer>            ::= <digits> [.]
<digits>             ::= {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
<mixed>              ::= [<digits>] . <digits>
<exponent>           ::= E [+|-] <digits>
<infinity>           ::= [+|-] INF
<NAN>                ::= NAN([<digits>])
```

Note: Square brackets enclose optional items, curly brackets enclose elements to be repeated at least once, and vertical bars separate alternative elements; letters that appear literally, like the 'E' marking the exponent field, may be either upper or lower case.

Table D-2
Syntax for String Conversions

Decform Records and Conversions from SANE types to Decimal Strings

Each conversion to a decimal string is controlled by a 'decform' record, which contains two fields:

```
style  -- 16-bit integer (0 or 1)
digits -- 16-bit integer
```

Style equals 0 for floating and 1 for fixed. Digits gives the number of significant digits for the floating style and the number of digits to the right of the decimal point for the fixed style (digits may be negative if the style is fixed). Decimal strings resulting from these conversions are always acceptable input for conversions from decimal strings to SANE types. Further formatting details are implementation dependent.

The Decimal Record Type

The decimal record type provides an intermediate unpacked form for programmers who wish to do their own parsing of numeric input or formatting of numeric output. The decimal record format has three fields:

```
sgn -- 16-bit integer (0 or 1)
exp -- 16-bit integer
sig -- string (maximum length is implementation-dependent)
```

The value represented is

$$(-1)^{\text{sgn}} * \text{sig} * 10^{\text{exp}}$$

when the length of sig is 18 or less. (Some implementations allow additional information in characters past the eighteenth.) Sig contains the integral decimal significand: the initial byte of sig (sig[0]) is the length byte, which gives the length of the ASCII string that is left-justified in the remaining bytes. Sgn is 0 for + and 1 for -. For example, if sgn = 1, exp = -3, and sig = '85' (sig[0] = 2, not shown), then the number represented is -0.085.

Conversions from Decimal Records to SANE Types

Conversions from the decimal record type handle any sig digit-string of length 18 or less (with an implicit decimal point at the right end). The following special cases apply:

- If sig[1] = '0' (zero), the decimal record is converted to zero. For example, a decimal record with sig = '0913' is converted to zero.
- If sig[1] = 'N', the decimal record is converted to a NaN. Except when the destination is of type comp (which has a unique NaN), the succeeding characters of sig are interpreted as a hex representation of the result significand: if fewer than 4 characters follow 'N' then they are right justified in the result's NaN code byte; if 4 or more characters follow 'N' then they are left justified in the result's significand; if no characters, or only '0's, follow N, then the result NaN code is set to nanzero = 15 (hex).
- If sig[1] = 'I' and the destination is not of comp type, the decimal record is converted to an infinity. If the destination is of comp type, the decimal record is converted to a NaN and invalid is signaled.
- Other cases produce undefined results.

Conversions from SANE Types to Decimal Records

Each conversion to a decimal record is controlled by a decform record (see above). All implementations allow at least 18 digits to be returned in sig. The implied decimal point is at the right end of sig, with exp set accordingly.

Zeroes, infinities, and NaNs are converted to decimal records with sig parts '0' (zero), 'I', and strings beginning with 'N', while exp is undefined. For NaNs, 'N' may be followed by a hex representation of the input significand. The third and forth hex digits following 'N' give the NaN code. For example, 'N0021000000000000' has NaN code 21 (hex).

When the number of digits specified in a decform record exceeds an implementation maximum (which is at least 18), the result is undefined.

A number may be too large to represent in a chosen fixed style. For instance, if the implementation's maximum length for sig is 18, then 10^{15} (which requires 16 digits to the left of the point in fixed-style representations) is too large for a fixed-style representation specifying more than 3 digits to the right of the point. If a number is too large for a chosen fixed style, then (depending on the SANE implementation) one of two results is returned: an implementation may return the most significant digits of the number in sig and set exp so that the decimal record

contains a valid floating-style representation of the number; alternatively, an implementation may simply set the string sig to '?'. Note that in any implementation, the test

```
(-exp <> decform digits) or (sig[1] = '?')
```

determines whether a nonzero finite number is too large for the chosen fixed style.

D.4.4 Conversions Between Decimal Formats

SANE implementations may provide conversions between decimal strings and decimal records.

Conversion from Decimal Strings to Decimal Records *Str 2 Dec - Sec 6-8*

This conversion routine is intended as an aid to programmers doing their own scanning. An integer argument on input gives the starting index into the string and on output is one greater than the index of the last character in the numeric sub-string just parsed. The longest possible numeric sub-string is parsed; if no numeric sub-string is recognized, then the index remains unchanged. Also, a Boolean argument is returned indicating that the input string, beginning at the input index, is a valid numeric string. The accepted input for this conversion is the same as for conversions from decimal string to SANE types (see above). Output is the same as for conversions from SANE types to decimal record (also above).

Examples

Input String	Index in out		Output value	Valid-prefix
12	1	3	12	TRUE
12E	1	3	12	TRUE
12E-	1	3	12	TRUE
12E-3	1	6	12E-3	TRUE
12E-x	1	3	12	FALSE
12E-3x	1	6	12E-3	FALSE
x12E-3	2	7	12E-3	TRUE
IN	1	1	UNDEFINED	TRUE
INF	1	4	INF	TRUE

Conversion from Decimal Records to Decimal Strings

This conversion is controlled by the style field of a decform record (the digits field is ignored). Input is the same as for conversions from decimal records to SANE types, and output formatting is the same as for conversions from SANE types to decimal strings. This conversion, actually a formatting operation, is exact and signals no exception.

D.5 Expression Evaluation

SANE arithmetic is extended-based. Arithmetic operations produce results with extended precision and extended range. For minimal loss of accuracy in more complicated computations, you should use extended temporary variables to store intermediate results.

D.5.1 Using Extended Temporaries

A programmer may use extended temporaries deliberately to reduce the effect of round-off error, overflow, and underflow on the final result.

Example 1

To compute the single-precision sum

$$S = X[1]*Y[1] + X[2]*Y[2] + \dots + X[N]*Y[N]$$

where X and Y are arrays of type single, declare an extended variable XS and compute

```
XS := 0;

for I := 1 to N do
    XS := XS + X[I]*Y[I]; {extended-precision arithmetic }
S := XS;                  {deliver final result to single.}
```

Even when input and output values have only single precision, it may be very difficult to prove that single-precision arithmetic is sufficient for a given calculation. Using extended-precision arithmetic for intermediate values will often improve the accuracy of single-precision results more than virtuoso algorithms would. Likewise, using the extra range of the extended type for intermediate results may yield correct final results in the single type in cases when using the single type for intermediate results would cause an overflow or a catastrophic underflow. Extended-precision arithmetic is also useful for calculations involving double or comp variables: see Example 2.

D.5.2 Extended-Precision Expression Evaluation

High-level languages that support SANE evaluate all non-integer numeric expressions to extended precision, regardless of the types of the operands.

Example 2

If C is of type comp and MAXCOMP is the largest comp value, then the right-hand side of

$$C := (\text{MAXCOMP} + \text{MAXCOMP}) / 2$$

would be evaluated in extended to the exact result $C = \text{MAXCOMP}$, even though the intermediate result $\text{MAXCOMP} + \text{MAXCOMP}$ exceeds the largest possible comp value.

D.5.3 Extended-Precision Expression Evaluation and the IEEE Standard

The IEEE Standard encourages extended-precision expression evaluation. Extended evaluation will on rare occasions produce results slightly different from those produced by other IEEE

implementations that lack extended evaluation. Thus in a single-only IEEE implementation,

$$z := x + y$$

with x , y , and z all single, is evaluated in one single-precision operation, with at most one rounding error. Under extended evaluation, however, the addition $x + y$ is performed in extended, then the result is coerced to the single precision of z , with at most two rounding errors. Both implementations conform to the standard.

The effect of a single- or double-only IEEE implementation can be obtained under SANE with rounding precision control, as described in Section D.8.

D.6 Comparisons

SANE supports the usual numeric comparisons: less, less-or-equal, greater, greater-or-equal, equal, and not-equal. For real numbers, these comparisons behave according to the familiar ordering of real numbers.

SANE comparisons handle NaNs and infinities as well as real numbers. The usual trichotomy for real numbers is extended so that, for any SANE values a and b , exactly one of the following is true:

$a < b$
 $a > b$
 $a = b$
 a and b are unordered

Determination is made by the rule:

If x or y is a NaN, then x and y are unordered; otherwise, x and y are less, equal, or greater according to the ordering of the real numbers, with the understanding that $+0 = -0 = \text{real } 0$, and $-\infty < \text{each real number} < +\infty$.

(Note that a NaN always compares unordered--even with itself.)

The meaning of high-level language relational operators is a natural extension of their old meaning based on trichotomy. For example, the Pascal or BASIC expression $x \leq y$ is true if x is less than y or if x equal y , and is false if x is greater than y or if x and y are unordered. Note that the SANE not-equal relation means less, greater, or unordered--even if not-equal is written $\langle \rangle$, as in Pascal and BASIC. High-level languages supporting SANE supplement the usual comparison operators by a function that takes two numeric arguments and returns the relation (less, equal, greater, or unordered) appropriate to its arguments. This function can be used to determine whether two numeric representations satisfy any combination of less, equal, greater, and unordered.

A high-level language comparison that involves a relational operator containing less or greater, but not unordered, signals invalid if the operands are unordered (that is, if either operand is a NaN). For example, in Pascal or BASIC if x or y is a quiet NaN then $x < y$, $x \leq y$, $x \geq y$, and $x > y$ signal invalid, but $x = y$ and $x \langle \rangle y$ (recall that $\langle \rangle$ contains unordered) do not. If a comparison operand is a signaling NaN, then invalid is signaled, just as in arithmetic operations.

D.7 Infinities, NaNs, and Denormalized Numbers

In addition to the normalized numbers supported by most floating-point packages, IEEE floating-point arithmetic also supports infinities, NaNs, and denormalized numbers.

D.7.1 Infinities

An *infinity* is a special bit pattern that can arise in one of two ways:

1. When a SANE operation should produce an exact mathematical infinity (such as $1/0$), the result is an infinity bit pattern.
2. When a SANE operation attempts to produce a number with magnitude too great for the number's intended floating-point storage format, the result may (depending on the current rounding direction) be an infinity bit pattern.

These bit patterns (as well as NaNs, introduced next) are recognized in subsequent operations and produce predictable results. The infinities, one positive (+INF) and one negative (-INF), generally behave as suggested by the theory of limits. For example, 1 added to +INF yields +INF; -1 divided by +0 yields -INF; and 1 divided by -INF yields -0.

Each of the storage types single, double, and extended provides unique representations for +INF and -INF. The comp type has no representations for infinities. (An infinity moved to the comp type becomes the comp NaN.)

D.7.2 NaNs

When a SANE operation cannot produce a meaningful result, the operation delivers a special bit pattern called a *NaN* (Not-a-Number). For example, 0 divided by 0, +INF added to -INF, and $\sqrt{-1}$ yield NaNs. A NaN can occur in any of the SANE storage types (single, double, extended, and comp); but, generally, system-specific integer types have no representation for NaNs. NaNs propagate through arithmetic operations. Thus, the result of 3.0 added to a NaN is the same NaN (that is, has the same NaN code). If two operands of an operation are NaNs, the result is one of the NaNs. NaNs are of two kinds: *quiet NaNs*, the usual kind produced by floating-point operations; and *signaling NaNs*. When a signaling NaN is encountered as an operand of an arithmetic operation, the invalid-operation exception is signaled and, if no halt occurs, a quiet NaN is produced for the result. Signaling NaNs could be used for uninitialized variables. They are not created by any SANE operations. The most significant bit of the field *f* illustrated in the section "Formats" in Section D.2 is clear for quiet NaNs and set for signaling NaNs. The unique comp NaN generally behaves like a quiet NaN.

A NaN in a floating-point format has an associated NaN code that indicates the NaN's origin. (These are listed in Table 7-1). The NaN code is the 8th through 15th most significant bits of the field *f* illustrated in Section D.2. The comp NaN is unique and has no NaN code.

$$\begin{aligned}
A_{22} &= A_{21}/2 = 0.000\ 0000\ 0000\ 0000\ 0000\ 0011 * 2^{-126} \\
A_{23} &= A_{22}/2 = 0.000\ 0000\ 0000\ 0000\ 0000\ 0010 * 2^{-126} \text{ (underflow)} \\
A_{24} &= A_{23}/2 = 0.000\ 0000\ 0000\ 0000\ 0000\ 0001 * 2^{-126} \\
A_{25} &= A_{24}/2 = 0.0 \text{ (underflow)}
\end{aligned}$$

$A_1 \dots A_{24}$ are denormalized; A_{24} is the smallest positive denormalized number.

Why Denormalized Numbers?

The use of denormalized numbers makes statements like the following true for all real numbers:

$$x - y = 0 \text{ if and only if } x = y$$

This statement is not true for most older systems of computer arithmetic, because they exclude denormalized numbers. For these systems, the smallest nonzero number is a normalized number with the minimum exponent; when the result of an operation is smaller than this smallest normalized number, the system delivers zero as the result. For such *flush-to-zero* systems, if $x \neq y$ but $x - y$ is smaller than the smallest normalized number, then $x - y = 0$. IEEE systems do not have this defect, as $x - y$, although denormalized, is not zero.

(A few old programs that rely on premature flushing to zero may require modification to work properly under IEEE arithmetic. For example, some programs may test $x - y = 0$ to determine whether x is very near y .)

D.7.4 Inquiries: Class and Sign

Each valid representation in a SANE data type (single, double, comp, or extended) belongs to exactly one of these classes:

- signaling NaN
- quiet NaN
- infinite
- zero
- normalized
- denormalized

SANE implementations provide the user with the facility to determine easily the class and sign of any valid representation.

D.8 Environmental Control

Environmental controls include the rounding direction, rounding precision, exception flags, and halt settings.

D.8.1 Rounding Direction

The available rounding directions are

- to-nearest
- upward
- downward
- toward-zero

The rounding direction affects all conversions and arithmetic operations except comparison and remainder. Except for conversions between binary and decimal (described in Section D.4), all operations are computed as if with infinite precision and range and then rounded to the destination format according to the current rounding direction. The rounding direction may be interrogated and set by the user.

The default rounding direction is to-nearest. In this direction the representable value nearest to the infinitely precise result is delivered; if the two nearest representable values are equally near, the one with least significant bit zero is delivered. Hence, halfway cases round to even when the destination is the comp or a system-specific integer type, and when the round-to-integer operation is used. If the magnitude of the infinitely precise result exceeds the format's largest value (by at least one half unit in the last place), then the corresponding signed infinity is delivered.

The other rounding directions are upward, downward, and toward-zero. When rounding upward, the result is the format's value (possibly INF) closest to and no less than the infinitely precise result. When rounding downward, the result is the format's value (possibly -INF) closest to and no greater than the infinitely precise result. When rounding toward zero, the result is the format's value closest to and no greater in magnitude than the infinitely precise result. To truncate a number to an integral value, use toward-zero rounding either with conversion into an integer format or with the round-to-integer operation.

D.8.2 Rounding Precision

Normally, SANE arithmetic computations produce results to extended precision and range. To facilitate simulations of arithmetic systems that are not extended-based, the IEEE Standard requires that the user be able to set the rounding precision to single or double. If the SANE user sets rounding precision to single (or double) then all arithmetic operations produce results that are correctly rounded and that overflow or underflow as if the destination were single (or double), even though results are typically delivered to extended formats. Conversions to double and extended formats are affected if rounding precision is set to single, and conversions to extended formats are affected if rounding precision is set to double; conversions to decimal, comp, and system-specific integer types are not affected by the rounding precision. Rounding precision can be interrogated as well as set.

Setting rounding precision to single or double does not provide a significant performance enhancement, and in some SANE implementations may hinder performance.

D.8.3 Exception Flags and Halts

SANE supports five exception flags with corresponding halt settings:

- invalid-operation (or invalid, for short)
- underflow

- overflow
- divide-by-zero
- inexact

These exceptions are signaled when detected; and, if the corresponding halt is enabled, the SANE engine will jump to a user-specified location. (A high-level language need not pass on to its user the facility to set this location, but may halt the user's program). The user's program can examine or set individual exception flags and halts, and can save and get the entire environment (rounding direction, rounding precision, exception flags, and halt settings). Further details of the halt (trap) mechanism are SANE implementation specific.

Exceptions

The *invalid-operation* exception is signaled if an operand is invalid for the operation to be performed. The result is a quiet NaN, provided the destination format is single, double, extended, or comp. The invalid conditions are these:

1. (addition or subtraction) magnitude subtraction of infinities, for example, $(+INF) + (-INF)$
2. (multiplication) $0 * INF$
3. (division) $0/0$ or INF/INF
4. (remainder) $x \text{ rem } y$, where y is zero or x is infinite
5. (square root) if the operand is less than zero
6. (conversion) to the comp format or to a system-specific integer format when excessive magnitude, infinity, or NaN precludes a faithful representation in that format (see Section D.4 for details)
7. (comparison) via predicates involving "<" or ">", but not "unordered", when at least one operand is a NaN
8. any operation on a signaling NaN except sign manipulations (negate, absolute-value, and copy-sign) and class and sign inquiries

The *divide-by-zero* exception is signaled when a finite nonzero number is divided by zero. It is also signaled, in the more general case, when an operation on finite operands produces an exact infinite result: for example, $\log_b(0)$ returns $-INF$ and signals divide-by-zero. (Overflow, rather than divide-by-zero, flags the production of an inexact infinite result.)

The *overflow* exception is signaled when a floating-point destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded. (Invalid, rather than overflow, flags the production of an out-of-range value for an integral destination format.)

The *underflow* exception is signaled when a floating-point result is both tiny and inexact (and therefore, perhaps significantly less accurate than it would be if the exponent range were unbounded). A result is considered tiny if, before rounding, its magnitude is smaller than its format's smallest positive normalized number.

The *inexact* exception is signaled if the rounded result of an operation is not identical to the mathematical (exact) result. Thus, *inexact* is always signaled in conjunction with overflow or underflow. Valid operations on infinities are always exact and therefore signal no exceptions. Invalid operations on infinities are described above.

D.8.4 Managing Environmental Settings

The environmental settings in SANE are global and can be explicitly changed by the user. Thus all routines inherit these settings and are capable of changing them. Often special precautions must be taken because a routine requires certain environment settings, or because a routine's settings are not intended to propagate outside the routine.

Example 1

The subroutine below uses to-nearest rounding while not affecting its caller's rounding direction. (Examples in this section use Pascal syntax. SANE implementations in other languages have operations with equivalent functionality.)

```
var
    r: RoundDir;           { local storage for rounding
                           direction}
...
begin
    r := GetRound;         { save caller's rounding direction }
    SetRound (ToNearest); { set to-nearest rounding }
    ...
    SetRound (r)           { restore caller's rounding
                           direction }
end;
```

Note that, if the subroutine is to be reentrant, then storage for the caller's environment must be local.

SANE implementations may provide two efficient functions for managing the environment as a whole: procedure-entry and procedure-exit.

The procedure-entry function returns the current environment (for saving in local storage) and sets the default environment: rounding direction to-nearest, rounding precision extended, and exception flags and halts clear.

Example 2

The following subroutine runs under the default environment while not affecting its caller's environment.

```

var
    e: Environment;      { local storage for environment }
...
begin
    ProcEntry (e);        { save caller's environment and }
                          { set default environment        }
    ...
    SetEnvironment (e) { restore caller's environment  }
end;

```

The procedure-exit function facilitates writing subroutines which appear to their callers to be atomic operations (such as addition, sqrt, and others). Atomic operations pass extra information back to their callers by signaling exceptions; however, they hide internal exceptions, which may be irrelevant or misleading. Procedure-exit, which takes an argument of the storage type for saving environments, does the following:

1. It temporarily saves the exception flags (raised by the subroutine).
2. It restores the environment received as argument.
3. It signals the temporarily saved exceptions. (This may cause a halt.)

Thus exceptions signaled between procedure-entry and procedure-exit are hidden from the calling program unless the exceptions remain raised when the procedure-exit function is called.

Example 3

The following function signals underflow if its result is denormal, and overflow if its result is infinite, but hides spurious exceptions occurring from internal computations.

```

function compres: double;
...
var e: Environment;      { local storage for environment}
    c: NumClass;          { for class inquiry          }
...
begin {compres}
    ProcEntry (e);        { save caller's environment and}
                          { and set default environment-}
                          { now halts disabled          }
    ...
    compres := result;     { result to be returned  }
    c := ClassDouble (result); { class inquiry      }
    SetException (-1, False); { clear possibly spurious
                              exceptions      }

    { now raise specified exception flags: }

```

```

if c = Infinite then
    SetException (Overflow, True)
else if c = DenormalNum then
    SetException (Underflow, True);

    ProcExit (e)      { restore caller's environment,      }
                      { including any halt enables, and }
                      { then signal exceptions from      }
                      { subroutine                        }

end {compres} ;

```

D.9 Auxiliary Procedures

SANE includes a set of special routines--

- negate
- absolute value
- copy-sign
- next-after
- scalb
- logb

--which are recommended in an appendix to the IEEE Standard as aids to programming.

D.9.1 Sign Manipulation

The sign manipulation operations change only the sign of their argument. Negate reverses the sign of its argument. Absolute-value makes the sign of its argument positive. Copy-sign takes two arguments and copies the sign of one of its arguments onto the sign of its other argument.

These operations are treated as nonarithmetic in the sense that they raise no exceptions: even signaling NaNs do not signal the invalid-operation exception.

D.9.2 Next-After Functions

The floating-point values representable in single, double, and extended formats constitute a finite set of real numbers. The next-after functions (one for each of the formats) generate the next representable neighbor in the proper format, given an initial value *x* and another value *y* indicating a direction from the initial value.

Each of the next-after functions takes two arguments, *x* and *y*:

```

NextSingle(x,y)   (x and y are single)
NextDouble(x,y)   (x and y are double)
NextExtended(x,y) (x and y are extended)

```

As elsewhere, the names of the functions may vary with the implementation.

Special Cases for Next-After Functions

- If the initial value and the direction value are equal, then the result is the initial value.
- If either argument is a quiet NaN, then the result is one or the other of the input NaNs.
- If the initial value is finite but the next representable number is infinite, then overflow and inexact are signaled.

If the next representable number lies strictly between $-M$ and $+M$, where M is the smallest positive normalized number for that format, and if the arguments are not equal, then underflow and inexact are signaled.

D.9.3 Binary Scale and Log Functions

The `scalb` and `logb` functions are provided for manipulating binary exponents.

`Scalb` efficiently scales a given number (x) by a given power (n) of 2, returning $x * 2^n$.

`Logb` returns the binary exponent of its input argument, $\log_b(x)$, as a signed integral value. When the input argument is denormalized, the exponent is determined as if the input argument had first been normalized.

Special Cases for Logb

- If x is infinite, `logb(x)` returns $+\text{INF}$.
- If $x = 0$, `logb(x)` returns $-\text{INF}$ and signals divide-by-zero.

D.10 The Elementary Functions

SANE provides a number of basic mathematical functions, including logarithms, exponentials, two important financial functions, trigonometric functions, and a random number generator. These functions are computed using the basic SANE arithmetic heretofore described.

All of the elementary functions, except the random number generator, handle NaNs, overflow, and underflow appropriately. All signal inexact appropriately, except that the general exponential and the financial functions may conservatively signal inexact when determining exactness would be too costly.

D.10.1 Logarithm Functions

SANE provides three logarithm functions.

- base-2 logarithm : $\log_2(x)$
- base-e or natural logarithm : $\ln(x)$
- base-e logarithm of argument plus 1 : $\ln1(x)$

$\text{Ln1}(x)$ accurately computes $\ln(x + 1)$. If the input argument x is small, such as an interest rate, the computation of $\text{Ln1}(x)$ is more accurate than the straightforward computation of $\ln(x + 1)$ by adding x to 1 and taking the natural logarithm of the result.

Special Cases for Logarithm Functions

- If $x = +\text{INF}$, then $\log_2(x)$, $\ln(x)$, and $\text{Ln1}(x)$ return $+\text{INF}$ and set no exceptions.
- If $x = 0$, then $\log_2(x)$ and $\ln(x)$ return $-\text{INF}$ and signal divide-by-zero. Similarly, if $x = -1$, then $\text{Ln1}(x)$ returns $-\text{INF}$ and signals divide-by-zero.
- If $x < 0$, then $\log_2(x)$ and $\ln(x)$ return a NaN and signal invalid. Similarly, if $x < -1$, then $\text{Ln1}(x)$ returns a NaN and signals invalid.

D.10.2 Exponential Functions

SANE provides five exponential functions.

- base-2 exponential : 2^x
- base-e or natural exponential : e^x
- base-e exponential minus 1 : $\text{exp1}(x)$
- integer exponential : x^i (i of integer type)
- general exponential : x^y

$\text{Exp1}(x)$ accurately computes $e^x - 1$. If the input argument x is small, such as an interest rate, then the computation of $\text{exp1}(x)$ is more accurate than the straightforward computation of $e^x - 1$ by exponentiation and subtraction.

Special Cases for 2^x , e^x , $\text{exp1}(x)$

- If $x = +\text{INF}$, then 2^x , e^x , and $\text{exp1}(x)$ return $+\text{INF}$. No exception is signaled.
- If $x = -\text{INF}$, then 2^x and e^x return 0; and $\text{exp1}(x)$ returns -1. No exception is signaled.

Special Cases for x^i

- If the integer exponent i equals 0 and x is not a NaN, then x^i returns 1. Note that with the integer exponential, $x^0 = 1$ even if x is zero or infinite.

Special Cases for x^y

- If x is $+0$ and y is negative, then the general exponential x^y returns $+\text{INF}$ and signals divide-by-zero.
- If x is -0 and y is integral and negative, then x^y returns $+\text{INF}$ if y is even, or $-\text{INF}$ if y is odd: both cases signal divide-by-zero.

The general exponential x^y returns a NaN and signals invalid if

- both x and y equal 0;
- $x = 1$ and y is infinite; or
- x is -0 or less than 0 and y is nonintegral.

D.10.3 Financial Functions

SANE provides two functions, *compound* and *annuity*, that can be used to solve various financial, or time-value-of-money, problems.

Compound

The *compound* function computes

$$\text{compound}(r, n) = (1 + r)^n$$

where r is the interest rate and n is the number (perhaps nonintegral) of periods. When the rate r is small, *compound* gives a more accurate computation than does the straightforward computation of $(1 + r)^n$ by addition and exponentiation.

Compound is directly applicable to computation of present and future values:

$$PV = FV * (1 + r)^{(-n)} = \frac{FV}{\text{compound}(r, n)}$$

$$FV = PV * (1 + r)^n = PV * \text{compound}(r, n)$$

Annuity

The *annuity* function computes

$$\text{annuity}(r, n) = \frac{1 - (1 + r)^{(-n)}}{r}$$

where r is the interest rate and n is the number of periods. *Annuity* is more accurate than the straightforward computation of the expression above using basic arithmetic operations and exponentiation. The *annuity* function is directly applicable to the computation of present and future values of ordinary annuities:

$$\begin{aligned} PV &= PMT * \frac{1 - (1 + r)^{(-n)}}{r} \\ &= PMT * \text{annuity}(r, n) \\ &\quad (1 + r)^n - 1 \end{aligned}$$

$$\begin{aligned}
FV &= PMT * \frac{1 - (1 + r)^{-n}}{r} \\
&= PMT * (1 + r)^n * \frac{1 - (1 + r)^{-n}}{r} \\
&= PMT * \text{compound}(r, n) * \text{annuity}(r, n)
\end{aligned}$$

where PMT is the amount of one periodic payment.

Special Cases for compound(r,n)

- If $r = 0$ and n is infinite, or if $r = -1$, then `compound(r,n)` returns a NaN and signals invalid.
- If $r = -1$ and $n < 0$, then `compound(r,n)` returns +INF and signals divide-by- zero.

Special Cases for annuity(r,n)

- If $r = 0$, then `annuity(r,n)` computes the sum of $1 + 1 + \dots + 1$ over n periods, and therefore returns the value n and signals no exceptions (the value n corresponds to the limit as r approaches 0).
- If $r < -1$, then `annuity(r,n)` returns a NaN and signals invalid.
- If $r = -1$ and $n > 0$, then `annuity(r,n)` returns -INF and signals divide-by-zero.

D.10.4 Trigonometric Functions

SANE provides the basic trigonometric functions

- cosine : `cos(x)`
- sine : `sin(x)`
- tangent : `tan(x)`
- arctangent : `arctan(x)`

The remaining trigonometric functions can be easily and efficiently computed from these four. The cosine, sine, and tangent functions use an argument reduction based on the remainder function (see Section D.3) and the nearest extended-precision approximation of $\pi/2$. Number results from arctangent lie between $-\pi/2$ and $\pi/2$. Thus the cosine, sine, and tangent functions have periods slightly different from their mathematical counterparts and diverge from their counterparts when their arguments become large.

Special Cases for sin(x), cos(x):

- If x is infinite, then `cos(x)` and `sin(x)` return a NaN and signal invalid.

Special Cases for tan(x):

- If x is the nearest extended approximation to $\pm\pi/2$, then tan(x) returns $\pm\text{INF}$.
- If x is infinite, then tan(x) returns a NaN and signals invalid.

Special Case for arctan(x):

- If $x = \pm\text{INF}$, then arctan(x) returns the nearest extended approximation to $\pm\pi/2$.

D.10.5 Random Number Generator

SANE provides a pseudorandom number generator, random. Random has one argument, passed by address. A sequence of (pseudo) random integral values r in the range

$$1 \leq r \leq 2^{31} - 2$$

can be generated by initializing an extended variable r to an integral value (the seed) in the above range and making repeated calls random(r); each call delivers in r the next random number in the sequence.

If seed values of r are non-integral or outside the range

$$1 \leq r \leq 2^{31} - 2$$

then results are unspecified.

A pseudorandom rectangular distribution on the interval (0,1) can be obtained by dividing the results from random by

$$2^{31} - 1 = \text{scalb}(31, 1) - 1 .$$

D.11 Annotated Bibliography

- [1] Apple Computer, Inc. "Appendix A: The Transcend and Realmodes Units" and "Appendix E: Floating-Point Arithmetic," *Apple III Pascal Programmer's Manual, Volume 2*, pp. 2-9, 56-85.

These appendixes describe the implementation of single-precision arithmetic in Apple III Pascal, which was based upon Draft 8.0 of the proposed Standard.

- [2] Apple Computer, Inc. *Apple III Pascal Numerics Manual: A Guide to Using the Apple III Pascal SANE and Elems Units*.

This manual describes the Apple III Pascal implementation of the Standard Apple Numeric Environment (SANE) through procedure calls to the SANE and Elems units. This was Apple's first full implementation of IEEE arithmetic.

- [3] Apple Computer, Inc. *Apple Pascal Numerics Manual: A Guide to Using the Apple Pascal SANE and Elems Units*.

This manual, generalized from [2], describes the Apple II and Apple III Pascal implementation of the Standard Apple Numeric Environment (SANE) through procedure calls to the SANE and Elems units.

- [4] Apple Computer, Inc. *The Apple Numerics Manual. Part I: The Standard Apple Numeric Environment. Part II: The 6502 Assembly-Language SANE Engine. Part III: The 68000 Assembly-Language SANE Engine*.

Part I describes the Standard Apple Numeric Environment, which applies to all Apple implementations of IEEE arithmetic: the appendix containing this bibliography is a reprint of Part I. Part II describes the 6502 assembly-language implementations used on Apple-II-series and Apple-III-series computers. Part III describes the 68000 assembly-language implementations used on Macintosh and Lisa.

- [5] Cody, W. J. "Analysis of Proposals for the Floating-Point Standard." *IEEE Computer*, Vol. 14, No. 3, March 1981, pp. 63-68.

This paper compares the several contending proposals presented to the Working Group.

- [6] Coonen, Jerome T. "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic." *IEEE Computer*, Vol. 13, No. 1, January 1980.

This paper is a forerunner to the work on the draft Standard.

- [7] Coonen, Jerome T. "Underflow and the Denormalized Numbers." *IEEE Computer*, Vol. 14, No. 3, March 1981, pp. 75-87.

- [8] Coonen, Jerome T. "Accurate, Yet Economical Binary-Decimal Conversions." To appear in *ACM Transactions on Mathematical Software*.

- [9] Demmel, James. "The Effects of Underflow on Numerical Computation." To appear in *SIAM Journal on Scientific and Statistical Computing*.

These papers examine one of the major features of the proposed Standard, gradual underflow, and show how problems of bounded exponent range can be handled through the use of denormalized values.

- [10] Fateman, Richard J. "High-Level Language Implications of the Proposed IEEE Floating-Point Standard." *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 2, April 1982, pp. 239-257.

This paper describes the significance to high-level languages, especially FORTRAN, of various features of the IEEE proposed Standard.

- [11] Floating-Point Working Group 754 of the Microprocessor Standards Committee, IEEE Computer Society. "A Standard for Binary Floating-Point Arithmetic." Proposed to IEEE, 345 East 47th Street, New York, NY 10017.

The implementation of SANE is based upon Draft 10.0 of this Standard.

- [12] Floating-Point Working Group 754 of the Microprocessor Standards Committee, IEEE Computer Society. "A Proposed Standard for Binary Floating-Point Arithmetic." *IEEE Computer*, Vol. 14, No. 3, March 1981, pp. 51-62.

This is Draft 8.0 of the proposed Standard, which was offered for public comment. The current Draft 10.0 is substantially simpler than this draft; for instance, warning mode and projective mode have been eliminated, and the definition of underflow has changed. However, the intent of the Standard is basically the same, and this paper includes some excellent introductory comments by David Stevenson, Chairman of the Floating-Point Working Group.

- [13] Hough, d. "Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic." *IEEE Computer*, Vol. 14, No. 3, March 1981, pp. 70-74.

This paper is an excellent introduction to the floating-point environment provided by the proposed Standard, showing how it facilitates the implementation of robust numerical computations.

- [14] Kahan, W. "Interval Arithmetic Options in the Proposed IEEE Floating-Point Arithmetic Standard," *Interval Mathematics 1980* (ed. K. E. L. Nickel). New York: Academic Press, New York, 1980, pp. 99-128.

This paper shows how the proposed Standard facilitates interval arithmetic.

- [15] Kahan, W., and Coonen, Jerome T. "The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments," *The Relationship between Numerical Computation and Programming Languages* (ed. J. K. Reid). New York: North Holland, 1982, pp. 103-115.

This paper describes high-level language issues relating to the proposed IEEE Standard, including expression evaluation and environment handling.

D.12 Glossary

Application type. A data type used to store data for applications.

Arithmetic type. A data type used to hold results of calculations inside the computer. The SANE arithmetic type, Extended, has greater range and precision than the application types, in order to improve the mathematical properties of the application types.

Binary floating-point number. A string of bits representing a sign, an exponent, and a significand. Its numerical value, if any, is the signed product of the significand and two raised to the power of its exponent.

Comp type. A 64-bit application data type for storing integral values of up to 19- or 20-decimal-digit precision. It is used for accounting applications, among others.

Denormalized number, or denorm. A nonzero binary floating-point number that is not normalized (that is, whose significand has a leading bit of zero) and whose exponent is the minimum exponent for the number's storage type.

Double type. A 64-bit application data type for storing floating-point values of up to 15- or 16-decimal-digit precision. It is used for statistical and financial applications, among others.

Environmental settings. The rounding direction, plus the exception flags and their respective halts.

Exceptions. Special cases, specified by the IEEE Standard, in arithmetic operations. The exceptions are INVALID, DIVBYZERO, OVERFLOW, UNDERFLOW, and INEXACT.

Exception flag. Each exception has a flag that can be set, cleared and tested. It is set when its respective exception occurs and stays set until explicitly cleared.

Exponent. The part of a binary floating-point number that indicates the power to which two is raised in determining the value of the number. The wider the exponent field in a numeric type, the greater range it will handle.

Extended type. An 80-bit arithmetic data type for storing floating-point values of up to 19- or 20-decimal-digit precision. SANE uses it to hold the results of arithmetic operations.

Halt. Each exception has a halt that can be set or cleared. If a halt is set, the program will halt when the exception occurs. Halts remain set until explicitly cleared.

Infinity. A special bit pattern produced when a floating-point operation attempts to produce a number greater in magnitude than the largest representable number in a given format. Infinities are signed.

Integer types. System types for integral values. Integer types typically use 16- or 32-bit 2's complement integers. Integer types are not SANE types but are available to SANE users.

Integral value. A mathematical integer: ..., -2, -1, 0, 1, 2,

NaN (Not a Number). A special bit pattern produced when a floating-point operation cannot produce a meaningful result (for example, 0/0 produces a NaN). NaNs can also be used for uninitialized storage. NaNs propagate through arithmetic operations.

Normalized number. A binary floating-point number in which all significand bits are significant: that is, the leading bit of the significand is 1.

Quiet NaN. A NaN that propagates through arithmetic operations without signaling an exception (and hence without halting a program).

Rounding direction. When the result of an arithmetic operation cannot be represented exactly in a SANE type, the computer must decide how to round the result. Under SANE, the computer resolves rounding decisions in one of four directions, chosen by the user: TONEAREST (the default), UPWARD, DOWNWARD, and TOWARDZERO.

Sign bit. The bit of a Single, Double, Comp, or Extended number that indicates the number's sign: 0 indicates a positive number; 1, a negative number.

Signaling NaN. A NaN that signals an INVALID exception when the NaN is an operand of an arithmetic operation. If no halt occurs, a quiet NaN is produced for the result. No SANE operation creates signaling NaNs.

Significand. The part of a binary floating-point number that indicates where the number falls between two successive powers of two. The wider the significand field in a numeric type, the more resolution it will have.

Single type. A 32-bit application data type for storing floating-point values of up to 7- or 8-decimal-digit precision. It is used for engineering applications, among others.

Appendix E

Interfaces to the Macintosh Toolbox and OS

This section lists all of the constants, types, variables, procedures, and functions comprising the Lightspeed Pascal interface to the Macintosh Toolbox and Operating System. Sections correspond to those in *Inside Macintosh*, and are in the following order:

AppleTalk Manager	E-3
Binary-Decimal Conversion Package	E-7
Control Manager	E-8
Desk Manager	E-10
Device and File Managers	E-11
Dialog Manager	E-17
Disk Driver	E-20
Disk Initialization Package	E-21
Event Manager (Toolbox)	E-22
Event Manager (OS)	E-24
Fixed-Point Math	E-25
Font Manager	E-26
General-Purpose Data Types	E-28
International Utilities Package	E-29
MacinTalk	E-32
Memory Manager	E-34

Memory Manager	E-34
Menu Manager	E-36
Package Manager	E-38
Printing Manager	E-39
QuickDraw	E-44
Resource Manager	E-52
SANE	E-54
Scrap Manager	E-57
Segment Loader	E-58
Serial Driver	E-59
Sound Drive	E-61
Standard File Package	E-63
System Error Handler	E-65
TextEdit	E-66
Three-Dimensional Graphics	E-68
Utilities (Toolbox)	E-70
Utilities (OS)	E-72
Vertical Retrace Manager	E-75
Window Manager	E-76

AppleTalk Manager

Constants

const

```
lapSize = 20;
ddpSize = 26;
nbpSize = 26;
atpSize = 56;

{error codes}

ddpSktErr      = -91;
ddpLenErr      = -92;
noBridgeErr    = -93;
LAPProtErr     = -94;
excessCollsns  = -95;

nbpBuffOvr     = -1024;
nbpNoConfirm   = -1025;
nbpConfDiff    = -1026;
nbpDuplicate   = -1027;
nbpNotFound    = -1028;
nbpNISErr      = -1029;

reqFailed      = -1096;
tooManyReqs    = -1097;
tooManySktS    = -1098;
badATPSkt      = -1099;
badBuffNum     = -1100;
noRelErr       = -1101;
cbNotFound     = -1102;
noSendResp     = -1103;
noDataArea     = -1104;
reqAborted     = -1105;

buf2SmallErr   = -3101;
noMPPerr       = -3102;
ckSumErr       = -3103;
extractErr     = -3104;
readQErr       = -3105;
atpLenErr      = -3106;
atpBadRsp      = -3107;
recNotFnd      = -3108;
sktClosedErr   = -3109;
```

Types

Note: Type BitMapType is changed to SignedByte since Lightspeed Pascal does not support bit-packing of Boolean array elements.

type

```
ABByte      = 1..127;

STR32       = string[32];

ABCallType  = (tLAPRead,tLAPWrite,tDDPRead,tDDPWrite,tNBPLookUp,
               tNBPCConfirm,tNBPCRegister,tATPSndRequest,
               tATPGetRequest,tATPSdRsp,tATPAddRsp,tATPRequest,
               tATPResponse);

ABProtoType = (lapProto,ddpProto,nbpProto,atpProto);

LAPAdrBlock = packed record
               dstNodeID:  Byte;
               srcNodeID:  Byte;
               LAPProtType: ABByte;
            end;

AddrBlock   = packed record
               aNet:      Integer;
               anode:     Byte;
               aSocket:   Byte;
            end;

EntityName  = record
               objStr:   Str32;
               typeStr:  Str32;
               zoneStr:  Str32;
            end;

EntityPtr   = ^EntityName;

RetransType = packed record
               retransInterval: Byte;
               retransCount:   Byte;
            end;

BitMapType  = SignedByte;

BDSElement = record
               BuffSize:  Integer;
               BuffPtr:   Ptr;
               DataSize:  Integer;
               UserBytes: LongInt;
            end;
```

```

BDSType      = array [0..7] of BDSElement;

BDSPtr       = ^BDSType;

ABusRecord   = record
    abOpCode:      abCallType;
    abResult:      Integer;
    abUserReference: LongInt;
    case abProtoType of
        lapProto: (
            lapAddress: LAPAdrBlock;
            lapReqCount: Integer;
            lapActCount: Integer;
            lapDataPtr: Ptr;
        );
        ddpProto: (
            ddpType:      Byte;
            ddpSocket:    Byte;
            ddpAddress:   AddrBlock;
            ddpReqCoun:   Integer;
            ddpActCount:  Integer;
            ddpDataPtr:   Ptr;
            ddpNodeID:    Byte;
        );
        nbpProto: (
            nbpEntityPtr: EntityPtr;
            nbpBufPtr:    Ptr;
            nbpBufSize:   Integer;
            nbpDataField: Integer;
            nbpAddress:   AddrBlock;
            nbpRetransmitInfo: RetransType;
        );
        atpProto: (
            atpSocket:    Byte;
            atpAddress:   AddrBlock;
            atpReqCount:  Integer;
            atpDataPtr:   Ptr;
            atpRspBDSPtr: BDSPtr;
            atpBitMap:    BitMapType;
            atpTransID:   Integer;
            atpActCount:  Integer;
            atpUserData:  LongInt;
            atpXO:        Boolean;
            atpEOM:       Boolean;
            atpTimeOut:   Byte;
            atpRetries:   Byte;
            atpNumBufs:   Byte;
            atpNumRsp:    Byte;
            atpBDSSize:   Byte;
            atpRspUData:  LongInt;
            atpRspBuf:    Ptr;
            atpRspSize:   Integer;
        );
    end; {record}

```

```

ABRecPtr      = ^ABusRecord;
ABRecHandle = ^ABRecPtr;

```

Routines

```

function LAPOpenProtocol (theLAPType: ABByte; protoPtr: Ptr): OsErr;
function LAPCloseProtocol (theLAPType: ABByte): OsErr;
function LAPRead          (abRecord: ABRecHandle; async: Boolean): OsErr;
function LAPWrite         (abRecord: ABRecHandle; async: Boolean): OsErr;
function LAPRdCancel      (abRecord: ABRecHandle): OsErr;

function DDPOpenSocket    (var theSocket: Byte;  sktListener: Ptr): OsErr;
function DDPCloseSocket   (theSocket: Byte): OsErr;
function DDPRead          (abRecord: ABRecHandle; retCksumErrs: Boolean;
                           async: Boolean): OsErr;
function DDPWrite         (abRecord: ABRecHandle; doCheckSum: Boolean;
                           async: Boolean): OsErr;
function DDPRdCancel      (abRecord: ABRecHandle): OsErr;

function NBPLoad:         OsErr;
function NBPUnLoad:       OsErr;
function NBPLookUp        (abRecord: ABRecHandle; async: Boolean): OsErr;
function NBPConfirm       (abRecord: ABRecHandle; async: Boolean): OsErr;
function NBPRegister      (abRecord: ABRecHandle; async: Boolean): OsErr;
function NBPRemove        (entityName: EntityPtr): OsErr;
function NBPExtract        (theBuffer: Ptr; numInBuf: Integer;
                           whichOne: Integer; var abEntity: EntityName;
                           var address: AddrBlock): OsErr;

function ATPLoad:         OsErr;
function ATPUnLoad:       OsErr;
function ATPOpenSocket    (addrRcvd: AddrBlock; var atpSocket: Byte):
                           OsErr;
function ATPCloseSocket   (atpSocket: Byte): OsErr;
function ATPSndRequest     (abRecord: ABRecHandle; async: Boolean): OsErr;
function ATPGetRequest     (abRecord: ABRecHandle; async: Boolean): OsErr;
function ATPSndRsp        (abRecord: ABRecHandle; async: Boolean): OsErr;
function ATPAddRsp        (abRecord: ABRecHandle): OsErr;
function ATPRequest       (abRecord: ABRecHandle; async: Boolean): OsErr;
function ATPResponse      (abRecord: ABRecHandle; async: Boolean): OsErr;
function ATPReqCancel     (abRecord: ABRecHandle; async: Boolean): OsErr;
function ATPRspCancel     (abRecord: ABRecHandle; async: Boolean): OsErr;

procedure RemoveHdlBlks;

function GetNodeAddress    (var myNode, myNet: Integer): OsErr;

function MPPOpen:         OsErr;
function MPPClose:        OsErr;
function IsMPPOpen:       Boolean;
function IsATPOpen:       Boolean;

```

Binary-Decimal Conversion Package

Routines

```
procedure StringToNum (theString: Str255; var theNum: LongInt);
```

```
procedure NumToString (theNum: LongInt; var theString: Str255);
```


Control Manager

Constants

const

```
{control messages}

drawCntl    = 0;
testCntl    = 1;
calcCRgns   = 2;
initCntl    = 3;
dispCntl    = 4;
posCntl     = 5;
thumbCntl   = 6;
dragCntl    = 7;
autoTrack   = 8;

{FindControl Result Codes}

inButton     = 10;
inCheckbox   = 11;
inUpButton   = 20;
inDownButton = 21;
inPageUp     = 22;
inPageDown   = 23;
inThumb      = 129;

{control definition proc ID's}

pushButProc  = 0;
checkBoxProc = 1;
radioButProc = 2;
scrollBarProc = 16;
useWFont     = 8;
```

Types

type

```
ControlHandle = ^ControlPtr;
ControlPtr    = ^ControlRecord;
```

```

ControlRecord = packed record
    nextControl:    ControlHandle;
    contrlOwner:    WindowPtr;
    contrlRect:     Rect;
    contrlVis:      Byte;
    contrlHilite:   Byte;
    contrlValue:    Integer;
    contrlMin:      Integer;
    contrlMax:      Integer;
    contrlDefProc:  Handle;
    contrlData:     Handle;
    contrlAction:   ProcPtr;
    contrlrfCon:    LongInt;
    contrlTitle:    Str255;
end; {ControlRecord}

```

Routines

```

function NewControl    (curWindow: windowPtr; boundsRect: Rect;
                        title: Str255; visible: Boolean; value: Integer;
                        min: Integer; max: Integer; contrlProc: Integer;
                        refCon: LongInt): ControlHandle;

procedure DisposeControl (theControl: ControlHandle);
procedure KillControls   (theWindow: WindowPtr);
procedure MoveControl    (theControl: ControlHandle; h,v: Integer);
procedure SizeControl    (theControl: ControlHandle; w,h: Integer);
procedure DragControl    (theControl: ControlHandle; startPt: Point;
                        bounds: Rect; slopRect: Rect; axis: Integer);

procedure ShowControl    (theControl: ControlHandle);
procedure HideControl    (theControl: ControlHandle);
procedure SetCTitle      (theControl: ControlHandle; title: Str255);
procedure GetCTitle      (theControl: ControlHandle; var title: Str255);
procedure HiliteControl  (theControl: ControlHandle; hiliteState: Integer);
procedure SetCRefCon     (theControl: ControlHandle; data: LongInt);
function GetCRefCon      (theControl: ControlHandle): LongInt;
procedure SetCtlValue    (theControl: ControlHandle; theValue: Integer);
function GetCtlValue     (theControl: ControlHandle): Integer;
function GetCtlMin       (theControl: ControlHandle): Integer;
function GetCtlMax       (theControl: ControlHandle): Integer;
procedure SetCtlMin      (theControl: ControlHandle; theValue: Integer);
procedure SetCtlMax      (theControl: ControlHandle; theValue: Integer);
function GetCtlAction    (theControl: ControlHandle): ProcPtr;
procedure SetCtlAction   (theControl: ControlHandle; newProc: ProcPtr);
function TestControl     (theControl: ControlHandle; thePt: Point): Integer;
function TrackControl    (theControl: ControlHandle; thePt: Point;
                        actionProc: ProcPtr): Integer;

function FindControl     (thePoint: Point; theWindow: WindowPtr;
                        var theControl: ControlHandle): Integer;

procedure DrawControls   (theWindow: WindowPtr);
function GetNewControl   (controlID: Integer;
                        owner: WindowPtr): ControlHandle;

```

Desk Manager

Routines

```
function  SystemEvent (myEvent: EventRecord): Boolean;

procedure SystemClick (theEvent: EventRecord; theWindow: windowPtr);

procedure SystemTask;

procedure SystemMenu (menuResult: LongInt);

function  SystemEdit (editCode: Integer): Boolean;

function  OpenDeskAcc (theAcc: Str255): Integer;

procedure CloseDeskAcc (refNum: Integer);
```

Device and File Managers

Constants

const

{finder constants}

```
fHasBundle    = 8192;
fInvisible    = 16384;
fTrash        = -3;
fDesktop      = -2;
fDisk         = 0;
```

{ioPosMode values}

```
fsAtMark      = 0;
fsFromStart   = 1;
fsFromLEOF    = 2;
fsFromMark    = 3;
rdVerify      = 64;
```

{ioPermission values}

```
fsCurPerm    = 0;
fsRdPerm      = 1;
fsWrPerm      = 2;
fsRdWrPerm    = 3;
```

{file system error codes}

```
DirFulErr     =   -33;    { Directory full }
DskFulErr     =   -34;    { disk full }
NSVErr        =   -35;    { no such volume }
IOErr         =   -36;    { I/O error }
BdNamErr      =   -37;    { bad name }
FNOpnErr      =   -38;    { File not open }
EOFErr        =   -39;    { end of file }
PosErr        =   -40;    { tried to position before start of file(r/w) }
MFulErr       =   -41;    { memory full (open) or file won't fit (load) }
TMFOErr       =   -42;    { too many files open }
FNFErr        =   -43;    { File not found }
WPrErr        =   -44;    { diskette is write protected }
FLckdErr      =   -45;    { file is locked }
VLckdErr      =   -46;    { volume is locked }
FBSyErr       =   -47;    { File is busy (delete) }
DupFNErr      =   -48;    { duplicate filename (rename) }
OpWrErr       =   -49;    { file already open with write permission}
ParamErr      =   -50;    { error in user parameter list }
RFNumErr      =   -51;    { refnum error }
GFPerr        =   -52;    { get file position error }
```

```

VolOffLinErr = -53;    { volume not on line error (was Ejected) }
PermErr      = -54;    { permissions error (on file open) }
VolOnLinErr  = -55;    { drive volume already on-line at MountVol }
NSDrvErr     = -56;    { no such drive (tried to mount bad drive num) }
NoMacDskErr  = -57;    { not a mac diskette (sig bytes are wrong) }
ExtFSErr     = -58;    { volume in question belongs to an external fs}
FSRnErr      = -59;    { file system rename error }
BadMDBErr    = -60;    { bad master directory block }
WrPermErr    = -61;    { write permissions error }
lastDskErr   = -64;    { last in a range of disk errors }
offLinErr    = -65;    { r/w requested for an off-line drive }
noNybErr     = -66;    { couldn't find 5 nybbles in 200 tries }
noAdrMkErr   = -67;    { couldn't find valid addr mark }
dataVerErr   = -68;    { read verify compare failed }
badCkSmErr   = -69;    { addr mark checksum didn't check }
badBtSlpErr  = -70;    { bad addr mark bit slip nibbles }
noDtaMkErr   = -71;    { couldn't find a data mark header }
badDCkSum    = -72;    { bad data mark checksum }
badDBtSlp    = -73;    { bad data mark bit slip nibbles }
wrUnderRun   = -74;    { write underrun occurred }
cantStepErr  = -75;    { step handshake failed }
tk0BadErr    = -76;    { track 0 detect doesn't change }
initIWMErr   = -77;    { unable to initialize IWM }
twoSideErr   = -78;    { tried to read 2nd side on a 1-sided drive }
spdAdjErr    = -79;    { unable to correctly adjust disk speed }
seekErr      = -80;    { track number wrong on address mark }
sectNFErr    = -81;    { sector number never found on a track }
firstDskErr  = -84;    { first in a range of disk errors }

```

Types

type

```
ParamBlkType = (IOParam, FileParam, VolumeParam, CntrlParam);
```

```
OSType = packed array[1..4] of Char;
```

```

FInfo = record                                {record of finder info}
    fdType:      OSType;  {the type of the file}
    fdCreator:   OSType;  {file's creator}
    fdFlags:     Integer; {flags hasbundle, invisible, locked, etc.}
    fdLocation:  Point;   {file's location in folder}
    fdFldr:     Integer;  {folder containing file}
end;                                {FInfo}

```

```
ParamBlockRec = record
```

```

    {12 byte header used by the file and IO system}
    qLink:      QElemPtr; {queue link in header}
    qType:      Integer;  {type byte for safety check}
    ioTrap:     Integer;  {FS: the Trap}
    ioCmdAddr:  Ptr;      {FS: address to dispatch to}

```

```
{common header to all variants}
ioCompletion: ProcPtr;    {completion routine addr (0 for synch calls)}
ioResult:      OSErr;      {result code}
ioNamePtr:     StringPtr;  {ptr to Vol:FileName string}
ioVRefNum:     Integer;    {volume refnum (DrvNum for Eject, MountVol)}
```

```
{different components for the different type of parameter blocks}
```

```
case ParamBlkType of
```

```
    ioParam: (
        ioRefNum:    Integer;    {refNum for I/O operation}
        ioVersNum:   SignedByte; {version number}
        ioPermsn:    SignedByte; {Open: permissions (byte)}
        ioMisc:      Ptr;        {Rename: new name}
                                   {GetEOF,SetEOF: logical end of file }
                                   {Open: optional ptr to buffer}
                                   {SetFileType: new type}
        ioBuffer:    Ptr;        {data buffer Ptr}
        ioReqCount:  LongInt;    {requested byte count}
        ioActCount:  LongInt;    {actual byte count completed}
        ioPosMode:   Integer;    {initial file positioning}
        ioPosOffset: LongInt;    {file position offset}
    );
```

```
    FileParam: (
        ioFRefNum:    Integer;    {reference number}
        ioFVersNum:   SignedByte; {version number}
        filler1:      SignedByte;
        ioFDirIndex:  Integer;    {GetFileInfo directory index}
        ioFlAttrib:   SignedByte; {GetFileInfo inuse bit=7, lock bit=0}
        ioFlVersNum:  SignedByte; {file version number}
        ioFlFndrInfo: FInfo;      {user info}
        ioFlNum:      LongInt;    {GetFileInfo: file number}
        ioFlStBlk:     Integer;    {start file block (0 if none)}
        ioFlLgLen:     LongInt;    {logical length (EOF)}
        ioFlPyLen:     LongInt;    {physical length}
        ioFlRStBlk:    Integer;    {start block rsrc fork}
        ioFlRLgLen:    LongInt;    {file logical length rsrc fork}
        ioFlRPyLen:    LongInt;    {file physical length rsrc fork}
        ioFlCrDat:     LongInt;    {file creation date & time
                                   (32 bits in secs)}
        ioFlMdDat:     LongInt;    {last modified date and time}
    );
```

```
    VolumeParam: (
        filler2:      LongInt;
        ioVolIndex:   Integer;    {volume index number}
        ioVCrDate:    LongInt;    {creation date and time}
        ioVLsBkUp:    LongInt;    {last backup date and time}
        ioVAttrb:     Integer;    {volume attrib}
        ioVNmFls:     Integer;    {number of files in directory}
        ioVDirSt:     Integer;    {start block of file directory}
        ioVBln:       Integer;    {GetVolInfo: length of dir in blocks}
        ioVNmAlBlks:  Integer;    {GetVolInfo: num blks (of alloc size)}
```

```

        ioValBlkSiz: LongInt;    {GetVolInfo: alloc blk byte size}
        ioVClpSiz:   LongInt;    {GetVolInfo: bytes to allocate at time}
        ioAlBlSt:    Integer;    {starting disk block in blockmap}
        ioVNxtFNum:  LongInt;    {GetVolInfo: next free file number}
        ioVFrBlk:    Integer;    {GetVolInfo: # free blks for this vol}
    );

    CntrlParam: (
        ioCRefNum: Integer        {refNum for I/O operation}
        CSCode:    Integer;       {word for control status code}
        CSParam:   array[0..10] of Integer; {operation-defined}
    );

end; {ParamBlockRec}

ParmBlkPtr = ^ParamBlockRec;

VCB = record
    qLink:      QElemPtr;  {link to next element}
    qType:      Integer;    {not used}
    vcbFlags:   Integer;
    vcbSigWord: Integer;
    vcbCrDate:  LongInt;
    vcbLsBkUp  LongInt;
    vcbAtrb:    Integer;
    vcbNmFls:   Integer;
    vcbDirSt:   Integer;
    vcbBlLn:    Integer;
    vcbNmBlks:  Integer;
    vcbAlBlkSiz: LongInt;
    vcbClpSiz:  LongInt;
    vcbAlBlSt:  Integer;
    vcbNxtFNum: LongInt;
    vcbFreeBks: Integer;
    vcbVN:      string[27];
    vcbDrvNum:  Integer;
    vcbDRefNum: Integer;
    vcbFSId:    Integer;
    vcbVRefNum: Integer;
    vcbMAdr:    Ptr;
    vcbBufAdr:  Ptr;
    vcbMLen:    Integer;
    vcbDirIndex: Integer;
    vcbDirBlk:  Integer;
end;

DctlEntry = record
    DctlDriver: Ptr;    {ptr to ROM, handle to RAM driver}
    DctlFlags:  Integer; {flags}
    DctlQHdr:   QHdr;   {driver's i/o queue}
    DctlPosition: LongInt; {byte pos used by read and write calls}
    DctlStorage: Handle;  {hndl to RAM drivers private storage}
    DctlRefNum:  Integer; {driver's reference number}
    DctlCurTicks: LongInt; {counter for timing system task calls}

```

```

        DCtlWindow:  Ptr;      {ptr to driver's window if any}
        DCtlDelay:   Integer; {number of ticks btwn sysTask calls}
        DCtlEMask:   Integer {desk accessory event mask}
        DCtlMenu:    Integer; {menu ID of menu associated with driver}
    end; {DCtlEntry}

    DCtlPtr = ^DCtlEntry;

    DCtlHandle = ^DCtlPtr;

{drive queue elements}
    DrvQEl = record
        qLink:      QElemPtr;
        qType:      Integer;
        dQDrive:    Integer;
        dQRefNum:   Integer; {ref num of the drvr which handles
                               this drive}
        dQFSID:     Integer; {id of file system which handles
                               this drive}
        dQDrvSize:  Integer; {size of drive in 512-byte blocks --
                               not for drvs 1&2}
    end;

    DrvQElPtr = ^DrvQEl;

```

Routines

```

procedure FInitQueue;
function GetFSQHdr: QHdrPtr;
function GetDrvQHdr: QHdrPtr;
function GetVCBQHdr: QHdrPtr;
function GetDCtlEntry (refNum: Integer): DCtlHandle;
function PBOpen (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBClose (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBRead (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBWrite (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBControl (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBStatus (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBKillIO (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBGetVInfo (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBGetVol (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBSetVol (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBFlushVol (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBCreate (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBDelete (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBOpenRF (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBRename (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBGetFInfo (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBSetFInfo (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBSetFLock (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBRstFLock (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;
function PBSetFVers (paramBlock: ParmBlkPtr; aSync: Boolean): OSErr;

```



```

function PBAllocate (paramBlock: ParmBlkPtr; aSync: Boolean): OSerr;
function PBGetEOF (paramBlock: ParmBlkPtr; aSync: Boolean): OSerr;
function PBSetEOF (paramBlock: ParmBlkPtr; aSync: Boolean): OSerr;
function PBGetFPos (paramBlock: ParmBlkPtr; aSync: Boolean): OSerr;
function PBSetFPos (paramBlock: ParmBlkPtr; aSync: Boolean): OSerr;
function PBFlushFile (paramBlock: ParmBlkPtr; aSync: Boolean): OSerr;
function PBMountVol (paramBlock: ParmBlkPtr): OSerr;
function PBUnMountVol (paramBlock: ParmBlkPtr): OSerr;
function PBEject (paramBlock: ParmBlkPtr): OSerr;
function PBOffLine (paramBlock: ParmBlkPtr): OSerr;
procedure AddDrive (drvRefNum: Integer; drvNum: Integer;
    QEl: drvQElPtr);

function FSOpen (fileName: Str255; vRefNum: Integer;
    var refNum: Integer): OSerr;

function FSClose (refNum: Integer): OSerr;
function FSRead (refNum: Integer; var count: LongInt;
    buffPtr: Ptr): OSerr;

function FSWrite (refNum: Integer; var count: LongInt;
    buffPtr: Ptr): OSerr;

function Control (refNum: Integer; csCode: Integer;
    csParamPtr: Ptr): OSerr;

function Status (refNum: Integer; csCode: Integer;
    csParamPtr: Ptr): OSerr;

function KillIO (refNum: Integer): OSerr;
function GetVInfo (drvNum: Integer; volName: StringPtr;
    var vRefNum: Integer;
    var FreeBytes: LongInt): OSerr;

function GetFInfo (fileName: Str255; vRefNum: Integer;
    var FndrInfo: FInfo): OSerr;

function GetVol (volName: StringPtr; var vRefNum: Integer): OSerr;
function SetVol (volName: StringPtr; vRefNum: Integer): OSerr;
function UnMountVol (volName: StringPtr; vRefNum: Integer): OSerr;
function Eject (volName: StringPtr; vRefNum: Integer): OSerr;
function FlushVol (volName: StringPtr; vRefNum: Integer): OSerr;
function Create (fileName: Str255; vRefNum: Integer;
    creator: OSType; fileType: OSType): OSerr;

function FSDelete (fileName: Str255; vRefNum: Integer): OSerr;
function OpenRF (fileName: Str255; vRefNum: Integer;
    var refNum: Integer): OSerr;

function Rename (oldName: Str255; vRefNum: Integer;
    newName: Str255): OSerr;

function SetFInfo (fileName: Str255; vRefNum: Integer;
    FndrInfo: FInfo): OSerr;

function SetFLock (fileName: Str255; vRefNum: Integer): OSerr;
function RstFLock (fileName: Str255; vRefNum: Integer): OSerr;
function Allocate (refNum: Integer; var count: LongInt): OSerr;
function GetEOF (refNum: Integer; var LogEOF: LongInt): OSerr;
function SetEOF (refNum: Integer; LogEOF: LongInt): OSerr;
function GetFPos (refNum: Integer; var filePos: LongInt): OSerr;
function SetFPos (refNum: Integer; posMode: Integer;
    posOff: LongInt): OSerr;

function GetVRefNum (fileRefNum: Integer; var vRefNum: Integer): OSerr;
function OpenDriver (name: Str255; var drvRefNum: Integer): OSerr;
function CloseDriver (refNum: Integer): OSerr;

```

Dialog Manager

Constants

const

```
userItem    = 0;
ctrlItem    = 4;
btnCtrl     = 0;           { Low two bits specify what kind of control }
chkCtrl     = 1;
radCtrl     = 2;
resCtrl     = 3;

statText    = 8;           { Static text  }
editText    = 16;          { Editable text  }
iconItem    = 32;          { Icon item  }
picItem     = 64;          { Picture item  }
itemDisable = 128;         { Disable item if set  }

OK          = 1;           { OK button is first by convention  }
Cancel      = 2;           { Cancel button is second by convention }
```

Types

Note: Type StageList is changed to Integer since Lightspeed Pascal does not support bit-packing of fields in a record.

type

```
DialogPtr    = WindowPtr;
DialogPeek   = ^DialogRecord;

DialogRecord = record
    window:    WindowRecord;
    Items:     Handle;
    textH:     TEHandle;
    EditField: Integer;
    EditOpen:  Integer;
    ADefItem:  Integer;
end;

DialogTHndl   = ^DialogTPtr;
DialogTPtr    = ^DialogTemplate;
```

```

DialogTemplate = record
    boundsRect: Rect;
    procID: Integer;
    visible: Boolean;
    filler1: Boolean;
    goAwayFlag: Boolean;
    filler2: Boolean;
    refCon: LongInt;
    itemsID: Integer;
    title: Str255;
end;

StageList = Integer;

AlertTPtr = ^AlertTemplate;

AlertTemplate = record
    boundsRect: Rect;
    itemsID: Integer;
    stages: StageList;
end;

```

Routines

```

procedure InitDialogs (resumeProc: ProcPtr);

function GetNewDialog (dialogID: Integer; wStorage: Ptr;
    behind: WindowPtr): DialogPtr;

function NewDialog (wStorage: Ptr; boundsRect: Rect; title: Str255;
    visible: Boolean; theProc: Integer;
    behind: WindowPtr; goAwayFlag: Boolean;
    refCon: LongInt; itmLstHndl: Handle): DialogPtr;

function IsDialogEvent (event: EventRecord): Boolean;

function DialogSelect (event: EventRecord; var theDialog: DialogPtr;
    var itemHit: Integer): Boolean;

procedure ModalDialog (filterProc: ProcPtr; var itemHit: Integer);

procedure DrawDialog (dialog: DialogPtr);

procedure CloseDialog (dialog: DialogPtr);

procedure DisposDialog (dialog: DialogPtr);

function Alert (alertID: Integer; filterProc: ProcPtr): Integer;

function StopAlert (alertID: Integer; filterProc: ProcPtr): Integer;

function NoteAlert (alertID: Integer; filterProc: ProcPtr): Integer;

```

```

function CautionAlert (alertID: Integer; filterProc: ProcPtr): Integer;

procedure CouldAlert      (alertID: Integer);

procedure FreeAlert       (alertID: Integer);

procedure CouldDialog     (DlgID: Integer);

procedure FreeDialog      (DlgID: Integer);

procedure ParamText       (cite0, cite1, cite2, cite3: Str255);

procedure ErrorSound      (sound: ProcPtr);

procedure GetDItem        (dialog: DialogPtr; itemNo: Integer;
                           var kind: Integer; var item: Handle;
                           var box: Rect);

procedure SetDItem        (dialog: DialogPtr; itemNo: Integer;
                           kind: Integer; item: Handle; box: Rect);

procedure SetIText        (item: Handle; text: Str255);

procedure GetIText        (item: Handle; var text: Str255);

procedure SelIText        (dialog: DialogPtr; itemNo: Integer;
                           startSel, endSel: Integer );

function GetAlrtStage: Integer;

procedure ResetAlrtStage;

procedure DlgCut          (dialog: DialogPtr);

procedure DlgPaste        (dialog: DialogPtr);

procedure DlgCopy         (dialog: DialogPtr);

procedure DlgDelete       (dialog: DialogPtr);

procedure SetDAFont       (fontNum: Integer);

```

Disk Driver

Types

type

```
DrvSts = record
    track:      Integer;      {current track}
    writeProt:  SignedByte;   {bit 7=1 if volume is locked}
    diskInPlace: SignedByte;   {disk in place}
    installed:  SignedByte;   {drive installed}
    sides:      SignedByte;   {bit 7=0 if single-sided drive}
    qLink:      QElemPtr;     {next queue entry}
    qType:      Integer;      {not used}
    dqDrive:    Integer;      {drive number}
    dqRefNum:   Integer;      {driver reference number}
    dqFSID:     Integer;      {file-system identifier}
    twoSideFmt: SignedByte;   {-1 if two-sided disk}
    needsFlush: SignedByte;   {reserved}
    diskErrs:   Integer;      {error count}
end; {DrvSts}
```

Routines

```
function DiskEject (drvnum: Integer): OSErr;

function SetTagBuffer (buffPtr: Ptr): OSErr;

function DriveStatus (drvNum: Integer; var status: DrvSts): OSErr;
```

Disk Initialization Package

Routines

procedure DILoad;

procedure DIUnLoad;

function DIBadMount (where: Point; evtMessage: LongInt): Integer;

function DIFormat (drvNum: Integer): OsErr;

function DIVERify (drvNum: Integer): OsErr;

function DIZero (drvNum: Integer; volName: Str255): OsErr;

Event Manager (Toolbox)

Constants

const

```
everyEvent    = -1;
NullEvent     = 0;
mouseDown     = 1;
mouseUp       = 2;
keyDown       = 3;
keyUp         = 4;
autoKey       = 5;
updateEvt     = 6;
diskEvt       = 7;
activateEvt   = 8;
networkEvt    = 10;
driverEvt     = 11;
app1Evt       = 12;
app2Evt       = 13;
app3Evt       = 14;
app4Evt       = 15;
```

{ event mask equates }

```
mDownMask     = 2;
mUpMask       = 4;
keyDownMask    = 8;
keyUpMask      = 16;
autoKeyMask    = 32;
updateMask     = 64;
diskMask       = 128;
activMask      = 256;
networkMask    = 1024;
driverMask     = 2048;
app1Mask       = 4096;
app2Mask       = 8192;
app3Mask       = 16384;
app4Mask       = -32768;
```

{to decipher event message for keyDown events}

```
charCodeMask = $000000FF;
keyCodeMask  = $0000FF00;
```

{ modifiers }

```
optionKey     = 2048; { Bit 3 of high byte }
alphaLock     = 1024; { Bit 2 }
ShiftKey      = 512;  { Bit 1 }
```

```

CmdKey      = 256;    { Bit 0 }
BtnState    = 128;    { Bit 7 of low byte is mouse button state }
activeFlag  = 1;      { bit 0 of modifiers for activate event }

```

Types

Note: Type KeyMap is changed from **packed array** [0..127] of Boolean to **array** [0..3] of Longint, since Lightspeed Pascal does not support bit-packing of Boolean array elements.

type

```

EventRecord = record
    what:      Integer;
    message:   LongInt;
    when:      LongInt;
    where:     Point;
    modifiers: Integer;
end;

KeyMap = array[0..3] of LongInt;

```

Routines

Note: Lightspeed Pascal, like Macintosh Pascal, allows the GetMouse procedure to optionally take two integer parameters (h, v) for the horizontal and vertical components of the mouse position.

```

function EventAvail (mask:Integer; var theEvent: EventRecord): Boolean;

function GetNextEvent (mask:Integer; var theEvent: EventRecord): Boolean;

function StillDown: Boolean;

function WaitMouseUp: Boolean;

procedure GetMouse (var pt: Point);

function TickCount: LongInt;

function Button: Boolean;

procedure GetKeys (var k: keyMap);

function GetDb1Time: LongInt;

function GetCaretTime: LongInt;

```


Event Manager (OS)

Constants

const

```
{error for PostEvent}  
EvtNotEnb = 1;
```

Types

type

```
evQEl = record  
    qLink:      QElemPtr;  
    qType:      Integer;  
    evtQwhat:    Integer;  
    evtQmessage: LongInt;  
    evtQwhen:    LongInt;  
    evtQwhere:   Point;  
    evtQmodifiers: Integer;  
end;
```

Routines

```
function PostEvent (eventNum: Integer; eventMsg: LongInt): OSErr;  
  
procedure FlushEvents (whichMask, stopMask: Integer);  
  
procedure SetEventMask (theMask: Integer);  
  
function OSEventAvail (mask: Integer; var theEvent: EventRecord): Boolean;  
  
function GetOSEvent (mask: Integer; var theEvent: EventRecord): Boolean;  
  
function GetEvQHdr: QHdrPtr;
```

Fixed-Point Math

Types

type

```
Fract = LongInt;
```

Routines

function FracMul (x, y: Fract): Fract;

{FracMul returns $x * y$. Note that FracMul effects "type * Fract --> type":

```
Fract   * Fract       --> Fract
LongInt * Fract       --> LongInt
Fract   * LongInt     --> LongInt
Fixed   * Fract       --> Fixed
Fract   * Fixed       --> Fixed}
```

function FixDiv (x, y: Fixed): Fixed;

{FixDiv returns x / y . Note that FixDiv effects "type / type --> Fixed":

```
Fixed / Fixed       --> Fixed
LongInt / LongInt   --> Fixed
Fract / Fract       --> Fixed
LongInt / Fixed     --> LongInt
Fract / Fixed       --> Fract}
```

function FracDiv (x, y: Fract): Fract;

{FracDiv returns x / y . Note that FracDiv effects "type / type --> Fract":

```
Fract / Fract       --> Fract
LongInt / LongInt   --> Fract
Fixed / Fixed       --> Fract
LongInt / Fract     --> LongInt
Fixed / Fract       --> Fixed}
```

function FracSqrt (x: Fract): Fract;

{FracSqrt returns the square root of x. Both argument and result are regarded as unsigned.}

function FracCos (x: Fixed): Fract;

function FracSin (x: Fixed): Fract;

{FracCos and FracSin return the cosine and sine, respectively, given the argument x in radians.}

Font Manager

Constants

const

```
commandMark    = $11;
checkMark      = $12;
diamondMark    = $13;
appleMark      = $14;

systemFont     = 0;
applFont       = 1;
newYork        = 2;
geneva         = 3;
monaco         = 4;
venice         = 5;
london         = 6;
athens         = 7;
sanFran       = 8;
toronto        = 9;
cairo          = 11;
losAngeles     = 12;
times          = 20;
helvetica      = 21;
courier        = 22;
symbol         = 23;
taliesin       = 24;

propFont       = $9000;
fixedFont      = $B000;
fontWid        = $ACB0;
```

Types

type

```
FMInput = packed record
    family: Integer;
    size: Integer;
    face: Style;
    needBits: Boolean;
    device: Integer;
    numer: Point;
    denom: Point;
end;

FMOutPtr = ^FMOutPut;
```

```

FMOutPut = packed record
    errNum:      Integer;
    fontHandle:  Handle;
    bold:        Byte;
    italic:       Byte;
    ulOffset:    Byte;
    ulShadow:    Byte;
    ulThick:     Byte;
    shadow:      Byte;
    extra:       SignedByte;
    ascent:      Byte;
    descent:     Byte;
    widMax:      Byte;
    leading:     SignedByte;
    unused:      Byte;
    numer:       Point;
    denom:       Point;
end;

FontRec = record
    fontType:    Integer; { font type }
    firstChar:   Integer; { ASCII code of first character }
    lastChar:    Integer; { ASCII code of last character }
    widMax:      Integer; { maximum character width }
    kernMax:     Integer; { negative of maximum character kern }
    nDescent:    Integer; { negative of descent }
    fRectWidth:  Integer; { width of font rectangle }
    fRectHeight: Integer; { height of font rectangle }
    owTLoc:      Integer; { offset to offset/width table }
    ascent:      Integer; { ascent }
    descent:     Integer; { descent }
    leading:     Integer; { leading }
    rowWords:    Integer; { row width of bit image / 2 }
    { bitImage:   array [1..rowWords,1..fRectHeight] of Integer;
      locTable:   array [firstChar..lastChar+2] of Integer;
      owTable:    array [firstChar..lastChar+2] of Integer; }
end;

```

Routines

```

procedure InitFonts;

procedure GetFontName (familyID: Integer; var theName: Str255);

procedure GetFNum (theName: Str255; var familyID: Integer);

procedure SetFontLock (lockFlag: Boolean);

function FMSwapFont (inRec: FMInput): FMOutPtr;

function RealFont (famID: Integer; size: Integer): Boolean;

```

General-Purpose Data Types

Types

type

```
SignedByte  = -128..127;      { any byte in memory }
Byte         = 0..255;        { unsigned byte for fontmgr }
Ptr          = ^SignedByte;   { blind pointer }
Handle       = ^Ptr;          { pointer to a master pointer }
ProcPtr      = Ptr;           { pointer to a procedure }
Fixed        = LongInt;       { fixed point arithmetic type }
Str255       = string[255];   { maximum string size }
StringPtr    = ^Str255;       { pointer to maximum string }
StringHandle = ^StringPtr;    { handle to maximum string }
```

International Utilities Package

Constants

const

```
{constants for manipulation of international resources}

currSymLead    = 16;  {set if currency symbol leads, reset if trails}
currNegSym     = 32;  {set if minus sign for neg num, reset if parens}
currTrailingZ  = 64;  {set if trailing zero}
currLeadingZ    = 128; {set if leading zero}

{constants specifying absolute value of short date form}

MDY = 0;  {month,day,year}
DMY = 1;  {day,month,year}
YMD = 2;  {year,month,day}

{masks used for date element format flags}

dayLdingZ = 32;  {set if leading zero for day}
mntLdingZ = 64;  {set if leading 0 for month}
century    = 128; {set if century, reset if no century}

{masks used for time element format flags}

secLeadingZ = 32;  {set if leading zero for seconds}
minLeadingZ = 64;  {set if leading zero for minutes}
hrLeadingZ  = 128; {set if leading zero for hours}

{country codes for version numbers}

verUS          = 0;
verFrance      = 1;
verBritain     = 2;
verGermany     = 3;
verItaly       = 4;
verNetherlands = 5;
verBelgiumLux  = 6;
verSweden      = 7;
verSpain       = 8;
verDenmark     = 9;
verPortugal    = 10;
verFrCanada    = 11;
verNorway      = 12;
verIsrael      = 13;
verJapan       = 14;
verAustralia   = 15;
verArabia      = 16;
verFinland     = 17;
```

```

verFrSwiss      = 18;
verGrSwiss      = 19;
verGreece       = 20;
verIceland      = 21;
verMalta        = 22;
verCyprus        = 23;
verTurkey       = 24;
verYugoslavia   = 25;

```

Types

type

```

intl0Hndl = ^intl0Ptr;
intl0Ptr  = ^intl0Rec;

intl0Rec  = packed record
    decimalPt: Char;    {ASCII Character for decimal point}
    thousSep:  Char;    {ASCII character for thousand separator}
    listSep:   Char;    {ASCII character for list separator}
    currSym1:  Char;    {ASCII for currency symbol (3 bytes long)}
    currSym2:  Char;
    currSym3:  Char;
    currFmt:   Byte;    {currency format flags}
    dateOrder: Byte;    {short date form - DMY,YMD, or MDY}
    shrtDateFmt: Byte;  {date elements format flag}
    dateSep:   Char;    {ASCII for date separator}
    timeCycle: Byte;    {indicates 12 or 24 hr cycle}
    timeFmt:   Byte;    {time elements format flags}
    mornStr:   packed array[1..4] of Char;
                    {ASCII for trailing string from 0:00 to 11:59}
    eveStr:    packed array[1..4] of Char;
                    {ASCII for trailing string from 12:00 to 23:59}
    timeSep:   Char;    {ASCII for the time separator}
    time1Suff: Char;    {suffix string used in 24 hr mode}
    time2Suff: Char;    {8 characters long}
    time3Suff: Char;
    time4Suff: Char;
    time5Suff: Char;
    time6Suff: Char;
    time7Suff: Char;
    time8Suff: Char;
    metricSys: Byte;    {indicates metric or English system}
    intl0Vers: Integer; {hi byte = country, low byte = vers}
end; {intl0Rec}

intl1Hndl = ^intl1Ptr;
intl1Ptr  = ^intl1Rec;

```

```

intl1Rec = packed record
    days:    array[1..7] of string[15];
              {length and word for Sunday through Monday}
    months:  array[1..12] of string[15];
              {length and word for January to December}
    suppressDay: Byte; {0 for day of week, 255 for no day of week}
    lngDateFmt: Byte; {expanded date format 0 or 255}
    dayLeading0: Byte; {255 for leading 0, 0 for no leading 0}
    abbrLen:   Byte; {length of abbreviated names in long form}
    st0: packed array[1..4] of Char;    {the string st0}
    st1: packed array[1..4] of Char;    {the string st1}
    st2: packed array[1..4] of Char;    {the string st2}
    st3: packed array[1..4] of Char;    {the string st3}
    st4: packed array[1..4] of Char;    {the string st4}
    intl1Vers: Integer; {version word}
    localRtn: Integer;  {routine for localizing mag comp;}
                      {minimal case is $4E75 for RTS, but
                      routine may be longer than one Integer.}

end; {intl1Rec}

DateForm = (shortDate, longDate, abbrevDate);

```

Routines

```

function IUGetIntl (theID: Integer): Handle;

procedure IUSetIntl (refNum: Integer; theID: Integer; intlParam: Handle);

procedure IUDateString (dateTime: LongInt; longFlag: DateForm;
    var result: Str255);

procedure IUDatePString (dateTime: LongInt; longFlag: DateForm;
    var result: Str255; intlParam: Handle);

procedure IUTimeString (dateTime: LongInt; wantSeconds: Boolean;
    var result: Str255);

procedure IUTimePString (dateTime: LongInt; wantSeconds: Boolean;
    var result: Str255; intlParam: Handle);

function IUMetric: Boolean;

function IUCompString (aStr,bStr: Str255): Integer;

function IUEqualString (aStr,bStr: Str255): Integer;

function IUMagString (aPtr,bPtr: Ptr; aLen,bLen: Integer): Integer;

function IUMagIDString (aPtr,bPtr: Ptr; aLen,bLen: Integer): Integer;

```


MacinTalk

Constants

const

```
noExcpsFile = '';           {signals Reader to use only basic rules}
noReader    = 'noReader';   {signals SpeechOn to not bring in Reader}
fullUnitT   = -4000;        {error code for driver unit table full}
```

Types

type

```
SpeechErr    = Integer;
SpeechRecord = array [0..99] of byte; {Driver parm block}
SpeechPointer = ^SpeechRecord;        {pointer to driver parm block}
SpeechHandle  = ^SpeechPointer;       {handle to driver parm block}

Sex          = (Male, Female);
F0Mode       = (Natural, Robotic, NoChange);
Language     = (xEnglish, French, Spanish, German, Italian);

VoiceRecord = record
    theSex:      Sex;
    theLanguage: Language;
    theRate:     Integer;
    thePitch:    Integer;
    theMode:     F0Mode;
    theName:     Str255;
    refCon:      LongInt;
end;

VoicePtr = ^VoiceRecord;
```

Routines

```
function SpeechOn (ExcpsFile: Str255;  
                   var theSpeech: SpeechHandle): SpeechErr;  
  
procedure SpeechOff (theSpeech: SpeechHandle);  
  
procedure SpeechRate (theSpeech: SpeechHandle; theRate: Integer);  
  
procedure SpeechPitch (theSpeech: SpeechHandle;  
                       thePitch: Integer; theMode: F0Mode);  
  
procedure SpeechSex (theSpeech: SpeechHandle; theSex: Sex);  
  
function Reader (theSpeech: SpeechHandle; EnglishInput: Ptr;  
                 InputLength: LongInt; PhoneticOutput: Handle): SpeechErr;  
  
function MacinTalk (theSpeech: SpeechHandle; Phonemes: Handle): SpeechErr;
```

Memory Manager

Constants

const

```
MemFullErr      = -108;    {Not enough room in heap zone}
NilHandleErr    = -109;    {Master Pointer was nil in HandleZone or other}
MemWZErr        = -111;    {WhichZone failed (applied to free block)}
MemPurErr       = -112;    {trying to purge locked or non-purgeable block}
MemLockedErr    = -117;    {Block is locked}
NoErr           = 0;       {All is well}
MaxSize         = $800000; {Max data block size is 512K bytes}
```

Types

type

```
Zone    = record
    BkLim:      Ptr;
    PurgePtr:   Ptr;
    HFstFree:   Ptr;
    ZCBFree:    LongInt;
    GZProc:     ProcPtr;
    MoreMast:   Integer;
    Flags:      Integer;
    CntRel:     Integer;
    MaxRel:     Integer;
    CntNRel:    Integer;
    MaxNRel:    Integer;
    CntEmpty:   Integer;
    CntHandles: Integer;
    MinCBFree:  LongInt;
    PurgeProc:  ProcPtr;
    SparePtr:   Ptr;           { reserved for future }
    AllocPtr:   Ptr;
    HeapData:   Integer;
end;

THz      = ^Zone;    { pointer to the start of a heap zone }

Size     = LongInt;  { size of a block in bytes }
```

Routines

```
procedure SetApplBase      (startPtr: Ptr);
procedure InitApplZone;
procedure InitZone        (pgrowZone: ProcPtr; cmoreMasters: Integer;
                           limitPtr, startPtr : Ptr);

function  GetZone:        THz;
procedure SetZone        (hz: THz);
function  ApplicZone:    THz;
function  SystemZone:    THz;
function  CompactMem     (cbNeeded: Size): Size;
procedure PurgeMem       (cbNeeded: Size);
function  FreeMem:       LongInt;
procedure ResrvMem       (cbNeeded: Size);
function  MaxMem         (var grow: Size): Size;
function  TopMem:        Ptr;
procedure SetGrowZone    (growZone: ProcPtr);
procedure SetApplLimit   (zoneLimit: Ptr);
function  GetApplLimit:  Ptr;
procedure MaxApplZone;
procedure MoveHHi        (h: handle);
function  NewPtr         (byteCount: Size): Ptr;
procedure DisposPtr      (p: Ptr);
function  GetPtrSize     (p: Ptr): Size;
procedure SetPtrSize     (p: Ptr; newSize: Size);
function  PtrZone        (p: Ptr): THz;
function  NewHandle      (byteCount: Size): Handle;
procedure DisposHandle   (h: Handle);
function  GetHandleSize  (h: Handle): Size;
procedure SetHandleSize  (h: Handle; newSize: Size);
function  HandleZone     (h: Handle): THz;
function  RecoverHandle  (p: Ptr): Handle;
procedure EmptyHandle    (h: Handle);
procedure ReAllocHandle  (h: Handle; byteCount: Size);
procedure HLock          (h: Handle);
procedure HUnLock        (h: Handle);
procedure HPurge         (h: Handle);
procedure HNoPurge       (h: Handle);
procedure MoreMasters;
procedure BlockMove      (srcPtr, destPtr: Ptr; byteCount: Size);
function  MemError:      OSErr;
function  GZSaveHnd:     Handle;
```

Menu Manager

Constants

const

```
noMark      = 0;    { mark symbol for MarkItem }
TextMenuProc = 0;

{ menu defProc messages }

mDrawMsg    = 0;
mChooseMsg  = 1;
mSizeMsg    = 2;
```

Types

type

```
MenuHandle = ^MenuPtr;
MenuPtr    = ^MenuInfo;

MenuInfo = record
    menuId:      Integer;
    menuWidth:   Integer;
    menuHeight:  Integer;
    menuProc:    Handle;
    enableFlags: LongInt;
    menuData:    Str255;
end;
```

Routines

```
procedure InitMenus;
function  NewMenu      (menuID: Integer; menuTitle: Str255): menuHandle;
function  GetMenu      (rsrcID: Integer): MenuHandle;
procedure DisposeMenu  (menu: menuHandle);
procedure AppendMenu   (menu: menuHandle; data: str255);
procedure InsertMenu   (menu: MenuHandle; beforeId: Integer);
procedure DeleteMenu   (menuId: Integer);
procedure DrawMenuBar;
procedure ClearMenuBar;
function  GetMenuBar:  Handle;
function  GetNewMBar   (menuBarID: Integer): Handle;
procedure SetMenuBar   (menuBar: Handle);
function  MenuSelect   (startPt: Point): LongInt;
function  MenuKey      (ch: Char): LongInt;
```

```

procedure HiLiteMenu      (menuId: Integer);
procedure SetItem          (menu: menuHandle; item: Integer;
                             itemString: Str255);
procedure GetItem          (menu: menuHandle; item: Integer;
                             var itemString: Str255);
procedure EnableItem       (menu: menuHandle; item: Integer);
procedure DisableItem      (menu: menuHandle; item: Integer);
procedure CheckItem        (menu: menuHandle; item: Integer; checked:
                             Boolean);
procedure SetItemIcon      (menu: menuHandle; item: Integer; iconNum: Byte);
procedure GetItemIcon      (menu: menuHandle; item: Integer;
                             var iconNum: Byte);
procedure SetItemStyle     (menu: menuHandle; item: Integer; styleVal: Style);
procedure GetItemStyle     (menu: menuHandle; item: Integer;
                             var styleVal: Style);
procedure SetItemMark      (menu: menuHandle; item: Integer; markChar: Char);
procedure GetItemMark      (menu: menuHandle; item: Integer;
                             var markChar: Char);
procedure SetMenuFlash     (flashCount: Integer);
procedure FlashMenuBar     (menuID: Integer);
function  GetMHandle       (menuID: Integer): menuHandle;
function  CountMItems      (menu: menuHandle): Integer;
procedure AddResMenu       (menu: menuHandle; theType:ResType);
procedure InsertResMenu    (menu: menuHandle; theType:ResType;
                             afterItem: Integer);
procedure CalcMenuSize     (menu:menuHandle);

```

Package Manager

Constants

```
const
    dskInit = 2;           {Disk Initializaton}
    stdFile = 3;           {Standard file}
    flPoint = 4;           {Floating-Point Arithmetic}
    trFunc  = 5;           {Transcendental Functions}
    intUtil = 6;           {International Utilities}
    bdConv  = 7;           {Binary/Decimal Conversion}
```

Routines

```
procedure InitAllPacks;

procedure InitPack (packID: Integer);
```

Printing Manager

Constants

const

```
{ Printing Methods }
bDraftLoop    = 0;
bSpoolLoop    = 1;
bUser1Loop    = 2;
bUser2Loop    = 3;

{ Printers }
bDevCItoh = 1;  iDevCItoh = $0100;  {CItoh}
bDevDaisy = 2;  iDevDaisy = $0200;  {Daisy}
bDevLaser = 3;  iDevLaser = $0300;  {Laser}

{ PrCtlCall parameters }
iPrBitsCtl    = 4;
lScreenBits   = $00000000;
lPaintBits    = $00000001;
lHiScreenBits = $00000010;
lHiPaintBits  = $00000011;
iPrIOCtl      = 5;
iPrEvtCtl     = 6;
lPrEvtAll     = $0002FFFD;
lPrEvtTop     = $0001FFFD;
iPrDevCtl     = 7;
lPrReset      = $00010000;
lPrPageEnd    = $00020000;
lPrLineFeed   = $00030000;
lPrLFSixth    = $0003FFFF;
lPrLFEighth   = $0003FFFE;
iFMgrCtl      = 8;

{ Result Codes }
iMemFullErr   = -108;
iPrAbort      = 128;
iIOAbort      = -27;
iPrSavPFI    = -1;

{ Miscellaneous }
sPrDrvr       = '.Print';
iPrDrvrRef    = -3;

iPrPgFract    = 120;
iPrPgFst      = 1;
iPrPgMax      = 9999;
iPrRelease    = 3;
iPfMaxPgs     = 128;
pPrGlobals    = $00000944;
```


Types

type

```
TPrVars = record
    iPrErr: Integer;
    bDocLoop: SignedByte;
    bUser1: SignedByte;
    lUser1: LongInt;
    lUser2: LongInt;
    lUser3: LongInt;
end;

TPPrVars = ^TPrVars;

TPrInfo = record
    iDev: Integer;
    iVRes: Integer;
    iHRes: Integer;
    rPage: Rect;
end;

TPPrInfo = ^TPrInfo;

TFeed = (feedCut, feedFanfold, feedMechCut, feedOther);

TPrStl = record
    wDev: Integer;
    iPageV: Integer;
    iPageH: Integer;
    bPort: SignedByte;
    feed: TFeed;
end;

TPPrStl = ^TPrStl;

TScan = (scanTB, scanBT, scanLR, scanRL);

TPrXInfo = record
    iRowBytes: Integer;
    iBandV: Integer;
    iBandH: Integer;
    iDevBytes: Integer;
    iBands: Integer;
    bPatScale: SignedByte;
    bULThick: SignedByte;
    bULOffset: SignedByte;
    bULShadow: SignedByte;
    scan: TScan;
    bXInfoX: SignedByte;
end;
```

```

TPPrXInfo = ^TPrXInfo;

TPrJob = record
    iFstPage: Integer;
    iLstPage: Integer;
    iCopies: Integer;
    bJDocLoop: SignedByte;
    fFromUsr: Boolean;
    pIdleProc: ProcPtr;
    pFileName: StringPtr;
    iFileVol: Integer;
    bFileVers: SignedByte;
    bJobX: SignedByte;
end;

TPPrJob = ^TPrJob;

TPrint = record
    iPrVersion: Integer;
    PrInfo: TPrInfo;
    rPaper: Rect;
    PrStl: TPrStl;
    PrInfoPT: TPrInfo;
    PrXInfo: TPrXInfo;
    PrJob: TPrJob;
    PrintX: array [1..19] of Integer;
end;

TPPrint = ^TPrint;
THPrint = ^TPPrint;

TPrPort = record
    GPort: GrafPort;
    GProcs: QDProcs;
    lGParam1: LongInt;
    lGParam2: LongInt;
    lGParam3: LongInt;
    lGParam4: LongInt;
    fOurPtr: Boolean;
    fOurBits: Boolean;
end;

TPPrPort = ^TPrPort;

TPrStatus = record
    iTotPages: Integer;
    iCurPage: Integer;
    iTotCopies: Integer;
    iCurCopy: Integer;
    iTotBands: Integer;
    iCurBand: Integer;
    fPgDirty: Boolean;
    fImaging: Boolean;
    hPrint: THPrint;

```

```

        pPrPort:    TPrPort;
        hPic:       PicHandle;
    end;

    TPrStatus = ^TPrStatus;

    TPfPgDir = record
        iPages:    Integer;
        lPgPos:    array [0..iPfMaxPgs] of Longint;
    end;

    TPPfPgDir = ^TPfPgDir;
    THPfPgDir = ^TPPfPgDir;

    TPfHeader = record
        Print:    TPrint;
        PfPgDir: TPfPgDir;
    end;

    TPPfHeader = ^TPfHeader;
    THPfHeader = ^TPPfHeader;

    TPrDlg = record
        Dlg:       DialogRecord;
        pFltrProc: ProcPtr;
        pItemProc: ProcPtr;
        hPrintUsr: THPrint;
        fDoIt:      Boolean;
        fDone:      Boolean;
        lUser1:     LongInt;
        lUser2:     LongInt;
        lUser3:     LongInt;
        lUser4:     LongInt;
        { ...Plus more stuff needed by the particular printing dialog... }
    end;

    TPPrDlg = ^TPrDlg;

```

Routines

```

{ Initialization }

procedure PrOpen;

procedure PrClose;

{ Print Dialogs & Default }

procedure PrintDefault (hPrint: THPrint);

function PrValidate (hPrint: THPrint): Boolean;

```

```

function PrStlDialog (hPrint: THPrint): Boolean;

function PrJobDialog (hPrint: THPrint): Boolean;

procedure PrJobMerge (hPrintSrc, hPrintDst: THPrint);

{ Document printing procs: These spool a print file. }

function PrOpenDoc (hPrint: THPrint; pPrPort: TPPrPort;
                    pIOBuf: Ptr): TPPrPort;

procedure PrCloseDoc (pPrPort: TPPrPort);

procedure PrOpenPage (pPrPort: TPPrPort; pPageFrame: TRect);

procedure PrClosePage (pPrPort: TPPrPort);

{ The "Printing Application" proc: Read and band the spooled PicFile. }

procedure PrPicFile (hPrint: THPrint; pPrPort: TPPrPort; pIOBuf: Ptr;
                    pDevBuf: Ptr; var PrStatus: TPrStatus);

{ Get/Set the current Print Error }

function PrError: Integer;

procedure PrSetError (iErr: Integer);

{ The .Print driver calls. }

procedure PrDrvrOpen;

procedure PrDrvrClose;

procedure PrCtlCall (iWhichCtl: Integer;
                    lParam1, lParam2, lParam3: LongInt);

{ Semi private stuff }

function PrStlInit (hPrint: THPrint): TPPrDlg;

function PrJobInit (hPrint: THPrint): TPPrDlg;

function PrDlgMain (hPrint: THPrint; pDlgInit: ProcPtr): Boolean;

procedure PrCfgDialog;

```

QuickDraw

Constants

const

```
{ the 16 transfer modes }

srcCopy      = 0;
srcOr        = 1;
srcXor       = 2;
srcBic       = 3;
notSrcCopy   = 4;
notSrcOr     = 5;
notSrcXor    = 6;
notSrcBic    = 7;
patCopy      = 8;
patOr        = 9;
patXor       = 10;
patBic       = 11;
notPatCopy   = 12;
notPatOr     = 13;
notPatXor    = 14;
notPatBic    = 15;

{ QuickDraw color separation constants }

normalBit    = 0;          { normal screen mapping  }
inverseBit   = 1;          { inverse screen mapping }
redBit       = 4;          { RGB additive mapping  }
greenBit     = 3;
blueBit      = 2;
cyanBit      = 8;          { CMYBk subtractive mapping }
magentaBit   = 7;
yellowBit    = 6;
blackBit     = 5;

{ colors expressed in these mappings }

blackColor   = 33;
whiteColor   = 30;
redColor     = 205;
greenColor   = 341;
blueColor    = 409;
cyanColor    = 273;
magentaColor = 137;
yellowColor  = 69;

{ standard picture comments }
picLParen    = 0;
picRParen    = 1;
```

Types

type

```
QDByte    = SignedByte;
QDPtr     = Ptr;          { blind pointer }
QDHandle  = Handle;       { blind handle  }
Pattern   = packed array [0..7] of 0..255;
Bits16    = array [0..15] of Integer;
VHSelect  = (v,h);
GrafVerb  = (frame,paint,erase,invert,fill);
StyleItem = (bold,italic,underline,outline,shadow,condense,extend);
Style     = set of StyleItem;

FontInfo  = record
    ascent: Integer;
    descent: Integer;
    widMax: Integer;
    leading: Integer;
end;

Point = record
    case Integer of
        0: (v: Integer; h: Integer);
        1: (vh: array[VHSelect] of Integer);
    end;

Rect = record
    case Integer of
        0: (top: Integer;
            left: Integer;
            bottom: Integer;
            right: Integer);
        1: (topLeft: Point;
            botRight: Point);
    end;

BitMap = record
    baseAddr: Ptr;
    rowBytes: Integer;
    bounds: Rect;
end;

Cursor = record
    data: Bits16;
    mask: Bits16;
    hotSpot: Point;
end;
```

```

PenState = record
    pnLoc:    Point;
    pnSize:   Point;
    pnMode:   Integer;
    pnPat:    Pattern;
end;

PolyHandle = ^PolyPtr;
PolyPtr    = ^Polygon;

Polygon    = record
    polySize:   Integer;
    polyBBox:   Rect;
    polyPoints: array[0..0] of Point;
end;

RgnHandle = ^RgnPtr;
RgnPtr    = ^Region;

Region    = record
    rgnSize:   Integer; { rgnSize = 10 for rectangular }
    rgnBBox:   Rect;
    { plus more data if not rectangular }
end;

PicHandle = ^PicPtr;
PicPtr    = ^Picture;

Picture    = record
    picSize:   Integer;
    picFrame:   Rect;
    { plus byte codes for picture content }
end;

QDProcsPtr = ^QDProcs;

QDProcs    = record
    textProc:   Ptr;
    lineProc:   Ptr;
    rectProc:   Ptr;
    rRectProc:  Ptr;
    ovalProc:   Ptr;
    arcProc:    Ptr;
    polyProc:   Ptr;
    rgnProc:    Ptr;
    bitsProc:   Ptr;
    commentProc: Ptr;
    txMeasProc: Ptr;
    getPicProc: Ptr;
    putPicProc: Ptr;
end;

GrafPtr    = ^GrafPort;

```

```

GrafPort = record
    device:      Integer;
    portBits:    BitMap;
    portRect:    Rect;
    visRgn:      RgnHandle;
    clipRgn:     RgnHandle;
    bkPat:       Pattern;
    fillPat:     Pattern;
    pnLoc:       Point;
    pnSize:      Point;
    pnMode:      Integer;
    pnPat:       Pattern;
    pnVis:       Integer;
    txFont:      Integer;
    txFace:      Style;
    txMode:      Integer;
    txSize:      Integer;
    spExtra:     Fixed;
    fgColor:     LongInt;
    bkColor:     LongInt;
    colrBit:     Integer;
    patStretch:  Integer;
    picSave:     Handle;
    rgnSave:     Handle;
    polySave:    Handle;
    grafProcs:   QDProcsPtr;
end;

```

Variables

var

```

thePort:  GrafPtr;
white:    Pattern;
black:    Pattern;
gray:     Pattern;
ltGray:   Pattern;
dkGray:   Pattern;
arrow:    Cursor;
screenBits: BitMap;
randSeed: LongInt;

```


Routines

Note: Lightspeed Pascal, like Macintosh Pascal, allows the Rect parameter in each of the following procedures to be alternatively specified as four Integer values giving the Top, Left, Bottom, and Right coordinates of the rectangle: FrameRect, PaintRect, EraseRect, InvertRect, FillRect, FrameOval, PaintOval, EraseOval, InvertOval, FillOval, FrameArc, PaintArc, EraseArc, InvertArc, FillArc, FrameRoundRect, PaintRoundRect, EraseRoundRect, InvertRoundRect, FillRoundRect.

{ GrafPort Routines }

```
procedure InitGraf      (globalPtr: Ptr);
procedure OpenPort      (port: GrafPtr);
procedure InitPort      (port: GrafPtr);
procedure ClosePort     (port: GrafPtr);
procedure SetPort       (port: GrafPtr);
procedure GetPort       (var port: GrafPtr);
procedure GrafDevice     (device: Integer);
procedure SetPortBits   (bm: BitMap);
procedure PortSize      (width,height: Integer);
procedure MovePortTo    (leftGlobal,topGlobal: Integer);
procedure SetOrigin     (h,v: Integer);
procedure SetClip       (rgn: RgnHandle);
procedure GetClip       (rgn: RgnHandle);
procedure ClipRect      (r: Rect);
procedure BackPat       (pat: Pattern);
```

{ Cursor Handling }

```
procedure InitCursor;
procedure SetCursor     (crsr: Cursor);
procedure HideCursor;
procedure ShowCursor;
procedure ObscureCursor;
```

{ Pen and Line Drawing }

```
procedure HidePen;
procedure ShowPen;
procedure GetPen        (var pt: Point);
procedure GetPenState   (var pnState: PenState);
procedure SetPenState   (pnState: PenState);
procedure PenSize       (width,height: Integer);
procedure PenMode       (mode: Integer);
procedure PenPat        (pat: Pattern);
procedure PenNormal;
procedure MoveTo        (h,v: Integer);
procedure Move          (dh,dv: Integer);
procedure LineTo        (h,v: Integer);
procedure Line          (dh,dv: Integer);
```

{ Text Drawing }

```
procedure TextFont      (font: Integer);
procedure TextFace      (face: Style);
procedure TextMode      (mode: Integer);
procedure TextSize      (size: Integer);
procedure SpaceExtra    (extra: Fixed);
procedure DrawChar      (ch: Char);
procedure DrawString    (s: Str255);
procedure DrawText      (textBuf: Ptr; firstByte,byteCount: Integer);
function CharWidth      (ch: Char): Integer;
function StringWidth    (s: Str255): Integer;
function TextWidth      (textBuf: Ptr; firstByte,byteCount: Integer):Integer;
procedure GetFontInfo   (var info: FontInfo);
```

{ Drawing in Color }

```
procedure ForeColor (color: LongInt);
procedure BackColor (color: LongInt);
procedure ColorBit   (whichBit: Integer);
```

{ Calculations with Points }

```
procedure AddPt      (src: Point; var dst: Point);
procedure SubPt      (src: Point; var dst: Point);
procedure SetPt      (var pt: Point; h,v: Integer);
function EqualPt     (pt1,pt2: Point): Boolean;
procedure ScalePt    (var pt: Point; fromRect,toRect: Rect);
procedure MapPt      (var pt: Point; fromRect,toRect: Rect);
procedure LocalToGlobal (var pt: Point);
procedure GlobalToLocal (var pt: Point);
```

{ Calculations with Rectangles }

```
procedure SetRect     (var r: Rect; left,top,right,bottom: Integer);
function EqualRect    (rect1,rect2: Rect): Boolean;
function EmptyRect    (r: Rect): Boolean;
procedure offsetRect  (var r: Rect; dh,dv: Integer);
procedure MapRect     (var r: Rect;      fromRect,toRect: Rect);
procedure InsetRect   (var r: Rect; dh,dv: Integer);
function SectRect     (src1,src2: Rect; var dstRect: Rect): Boolean;
procedure UnionRect   (src1,src2: Rect; var dstRect: Rect);
function PtInRect     (pt: Point; r: Rect): Boolean;
procedure Pt2Rect     (pt1,pt2: Point; var dstRect: Rect);
```

{ Graphical Operations on Rectangles }

```
procedure FrameRect   (r: Rect);
procedure PaintRect   (r: Rect);
procedure EraseRect   (r: Rect);
procedure InvertRect  (r: Rect);
procedure FillRect    (r: Rect; pat: Pattern);
```

{ Graphical Operations on Rounded-Corner Rectangles }

```
procedure FrameRoundRect (r: Rect; ovWd,ovHt: Integer);
procedure PaintRoundRect (r: Rect; ovWd,ovHt: Integer);
procedure EraseRoundRect (r: Rect; ovWd,ovHt: Integer);
procedure InvertRoundRect (r: Rect; ovWd,ovHt: Integer);
procedure FillRoundRect (r: Rect; ovWd,ovHt: Integer; pat: Pattern);
```

{ Graphical Operations on Ovals }

```
procedure FrameOval (r: Rect);
procedure PaintOval (r: Rect);
procedure EraseOval (r: Rect);
procedure InvertOval (r: Rect);
procedure FillOval (r: Rect; pat: Pattern);
```

{ Graphical Operations on Arcs }

```
procedure FrameArc (r: Rect; startAngle,arcAngle: Integer);
procedure PaintArc (r: Rect; startAngle,arcAngle: Integer);
procedure EraseArc (r: Rect; startAngle,arcAngle: Integer);
procedure InvertArc (r: Rect; startAngle,arcAngle: Integer);
procedure FillArc (r: Rect; startAngle,arcAngle: Integer; pat: Pattern);
procedure PtToAngle (r: Rect; pt: Point; var angle: Integer);
```

{ Graphical Operations on Polygons }

```
function OpenPoly: PolyHandle;
procedure ClosePoly;
procedure KillPoly (poly: PolyHandle);
procedure offsetPoly (poly: PolyHandle; dh,dv: Integer);
procedure MapPoly (poly: PolyHandle; fromRect,toRect: Rect);
procedure FramePoly (poly: PolyHandle);
procedure PaintPoly (poly: PolyHandle);
procedure ErasePoly (poly: PolyHandle);
procedure InvertPoly (poly: PolyHandle);
procedure FillPoly (poly: PolyHandle; pat: Pattern);
```

{ Calculations with Regions }

```
function NewRgn: RgnHandle;
procedure DisposeRgn (rgn: RgnHandle);
procedure CopyRgn (srcRgn,dstRgn: RgnHandle);
procedure SetEmptyRgn (rgn: RgnHandle);
procedure SetRectRgn (rgn: RgnHandle; left,top,right,bottom: Integer);
procedure RectRgn (rgn: RgnHandle; r: Rect);
procedure OpenRgn;
procedure CloseRgn (dstRgn: RgnHandle);
procedure offsetRgn (rgn: RgnHandle; dh,dv: Integer);
procedure MapRgn (rgn: RgnHandle; fromRect,toRect: Rect);
procedure InsetRgn (rgn: RgnHandle; dh,dv: Integer);
procedure SectRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
procedure UnionRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
procedure DiffRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
```

```

procedure XorRgn      (srcRgnA,srcRgnB,dstRgn: RgnHandle);
function  EqualRgn    (rgnA,rgnB: RgnHandle): Boolean;
function  EmptyRgn    (rgn: RgnHandle): Boolean;
function  PtInRgn     (pt: Point; rgn: RgnHandle): Boolean;
function  RectInRgn   (r: Rect; rgn: RgnHandle): Boolean;

{ Graphical Operations on Regions }

procedure FrameRgn   (rgn: RgnHandle);
procedure PaintRgn   (rgn: RgnHandle);
procedure EraseRgn   (rgn: RgnHandle);
procedure InvertRgn  (rgn: RgnHandle);
procedure FillRgn    (rgn: RgnHandle; pat: Pattern);

{ Graphical Operations on BitMaps }

procedure ScrollRect (dstRect: Rect; dh,dv: Integer; updateRgn: rgnHandle);
procedure CopyBits   (srcBits,dstBits: BitMap; srcRect,dstRect: Rect;
                      mode: Integer; maskRgn: RgnHandle);

{ Pictures }

function  OpenPicture (picFrame: Rect): PicHandle;
procedure ClosePicture;
procedure DrawPicture (myPicture: PicHandle; dstRect: Rect);
procedure PicComment  (kind,dataSize: Integer; dataHandle: Handle);
procedure KillPicture (myPicture: PicHandle);

{ The Bottleneck Interface: }

procedure SetStdProcs (var procs: QDProcs);
procedure StdText     (count: Integer; textAddr: Ptr; numer,denom: Point);
procedure StdLine     (newPt: Point);
procedure StdRect     (verb: GrafVerb; r: Rect);
procedure StdRRRect   (verb: GrafVerb; r: Rect; ovWd,ovHt: Integer);
procedure StdOval     (verb: GrafVerb; r: Rect);
procedure StdArc      (verb: GrafVerb; r: Rect;
                      startAngle,arcAngle: Integer);
procedure StdPoly     (verb: GrafVerb; poly: PolyHandle);
procedure StdRgn      (verb: GrafVerb; rgn: RgnHandle);
procedure StdBits     (var srcBits: BitMap; var srcRect,dstRect: Rect;
                      mode: Integer; maskRgn: RgnHandle);
procedure StdComment  (kind,dataSize: Integer; dataHandle: Handle);
function  StdTxMeas   (count: Integer; textAddr: Ptr;
                      var numer,denom: Point;
                      var info: FontInfo): Integer;
procedure StdGetPic   (dataPtr: Ptr; byteCount: Integer);
procedure StdPutPic   (dataPtr: Ptr; byteCount: Integer);

{ Miscellaneous Utility Routines }

function  GetPixel    (h,v: Integer): Boolean;
function  Random:     Integer;
procedure StuffHex    (thingptr: Ptr; s:Str255);

```

Resource Manager

Constants

const

```
{Resource attribute byte}
resSysHeap    = 64;    { System or application heap? }
resPurgeable  = 32;    { Purgeable resource? }
resLocked     = 16;    { Load it in locked? }
resProtected  = 8;     { Protected? }
resPreload    = 4;     { Load in on OpenResFile? }
resChanged    = 2;     { Resource changed? }

mapReadOnly   = 128;   { Resource file read-only }
mapCompact    = 64;    { Compact resource file }
mapChanged    = 32;    { Write map out at update }

resNotFound   = -192;  { Resource not found }
resFNotFound   = -193;  { Resource file not found }
addResFailed  = -194;  { AddResource failed }
rmvResFailed  = -196;  { RmveResource failed }

{ID's for resources provided in sysResDef}

{standard cursor definitions}
iBeamCursor = 1;      {text selection cursor}
crossCursor  = 2;      {for drawing graphics}
plusCursor   = 3;      {for structured selection}
watchCursor  = 4;      {for indicating a long delay}

{icons}
stopIcon     = 0;
noteIcon     = 1;
cautionIcon = 2;

{patterns}
sysPatListID = 0;     {ID of PAT# which contains 38 patterns}
```

Types

type

```
ResType = packed array [1..4] of Char;
```

Routines

```
function InitResources: Integer;
procedure RsrcZoneInit;
procedure CreateResFile (fileName: Str255);
function OpenResFile (fileName: Str255): Integer;
procedure UseResFile (refNum: Integer);
function GetResFileAttrs (refNum: Integer): Integer;
procedure SetResFileAttrs (refNum: Integer; attrs: Integer);
procedure UpdateResFile (refNum: Integer);
procedure CloseResFile (refNum: Integer);
procedure SetResPurge (install: Boolean);
procedure SetResLoad (AutoLoad: Boolean);
function CountResources (theType: ResType): Integer;
function GetIndResource (theType: ResType; index: Integer): Handle;
function CountTypes: Integer;
procedure GetIndType (var theType: ResType; index: Integer);
function UniqueID (theType: ResType): Integer;
function GetResource (theType: ResType; ID: Integer): Handle;
function GetNamedResource (theType: ResType; name: Str255): Handle;
procedure LoadResource (theResource: Handle);
procedure ReleaseResource (theResource: Handle);
procedure DetachResource (theResource: Handle);
procedure ChangedResource (theResource: Handle);
procedure WriteResource (theResource: Handle);
function HomeResFile (theResource: Handle): Integer;
function CurResFile: Integer;
function GetResAttrs (theResource: Handle): Integer;
procedure SetResAttrs (theResource: Handle; attrs: Integer);
procedure GetResInfo (theResource: Handle;
    var theID: Integer;
    var theType: ResType;
    var name: Str255);
procedure SetResInfo (theResource: Handle; theID: Integer;
    name: Str255);
procedure AddResource (theResource: Handle; theType: ResType;
    theID: Integer; name: Str255);
procedure RmveResource (theResource: Handle);
function SizeResource (theResource: Handle): LongInt;
function ResError: Integer;
```

SANE

Constants

const

```
DecStrLen = 255;
SigDigLen = 20;

{ Exceptions }

Invalid      = 1;
Underflow    = 2;
Overflow     = 4;
DivByZero    = 8;
Inexact      = 16;
```

Types

type

```
{ Types for handling decimal representations }

DecStr      = string[DecStrLen];

CStrPtr     = ^Char;

Decimal     = record
                sgn : 0..1;
                exp : Integer;
                sig : string[SigDigLen]
            end;

DecForm     = record
                style : (FloatDecimal, FixedDecimal);
                digits : Integer;
            end;

{ Ordering relations }

RelOp       = (GreaterThan, LessThan, EqualTo, Unordered);

{ Inquiry classes }

NumClass    = (SNaN, QNaN, Infinite, ZeroNum, NormalNum, DenormalNum);

{ Environmental control }

Exception   = Integer;
```

```

RoundDir      = (ToNearest, Upward, Downward, TowardZero);

RoundPre      = (ExtPrecision, DblPrecision, RealPrecision);

Environment   = Integer;

```

Routines

```

{ Conversions between numeric binary types }

function Num2Integer ( x : extended ) : Integer;
function Num2Longint ( x : extended ) : LongInt;
function Num2Real     ( x : extended ) : real;
function Num2Double   ( x : extended ) : double;
function Num2Extended ( x : extended ) : extended;
function Num2Comp     ( x : extended ) : comp;

{ Conversions between binary and decimal }

procedure Num2Dec ( f : DecForm; x : extended; var d : Decimal );
function Dec2Num ( d : Decimal ) : extended;
procedure Num2Str ( f : DecForm; x : extended; var s : DecStr );
function Str2Num ( s : DecStr ) : extended;

{ Conversions between decimal formats }

procedure Str2Dec ( s : DecStr; var Index : Integer;
                  var d : Decimal; var ValidPrefix : Boolean );
procedure CStr2Dec ( s : CStrPtr; var Index : Integer;
                   var d : Decimal; var ValidPrefix : Boolean );
procedure Dec2Str ( f : DecForm; d : Decimal; var s : DecStr );

{ Arithmetic, auxiliary, and elementary functions }

function Remainder ( x, y : extended; var quo : Integer ) : extended;
function Rint      ( x : extended ) : extended;
function Scalb     ( n : Integer; x : extended ) : extended;
function Logb      ( x : extended ) : extended;
function CopySign  ( x, y : extended ) : extended;

function NextReal   ( x, y : real ) : real;
function NextDouble ( x, y : double ) : double;
function NextExtended ( x, y : extended ) : extended;

function Log2      ( x : extended ) : extended;
function Ln1       ( x : extended ) : extended;
function Exp2      ( x : extended ) : extended;
function Exp1      ( x : extended ) : extended;
function XpwrI     ( x : extended; i : Integer ) : extended;
function XpwrY     ( x, y : extended ) : extended;

```



```

function Compound ( r, n : extended ) : extended;
function Annuity ( r, n : extended ) : extended;

function Tan ( x : extended ) : extended;
function RandomX ( var x : extended ) : extended;

{ Inquiry routines }

function ClassReal ( x : real ) : NumClass;
function ClassDouble ( x : double ) : NumClass;
function ClassComp ( x : comp ) : NumClass;
function ClassExtended ( x : extended ) : NumClass;
function SignNum ( x : extended ) : Integer;

{ NaN function }

function NAN ( i : Integer ) : extended;

{ Environment access routines }

procedure SetException ( e : Exception; b : Boolean );
function TestException ( e : Exception ) : Boolean;
procedure SetHalt ( e : Exception; b : Boolean );
function TestHalt ( e : Exception ) : Boolean;
procedure SetRound ( r : RoundDir );
function GetRound : RoundDir;
procedure SetPrecision ( p : RoundPre );
function GetPrecision : RoundPre;
procedure SetEnvironment ( e : Environment );
procedure GetEnvironment ( var e : Environment );
procedure ProcEntry ( var e : Environment );
procedure ProcExit ( e : Environment );
function GetHaltVector : LongInt;
procedure SetHaltVector ( v : LongInt );

{ Comparison routine }

function Relation ( x, y : extended ) : Relop;

```

Scrap Manager

Constants

const

```
noScrapErr = -100;  {desk scrap isn't initialized}
noTypeErr  = -102;
```

Types

type

```
ScrapStuff = record
    scrapSize:   LongInt;
    scrapHandle: Handle;
    scrapCount:  Integer;
    scrapState:  Integer;
    scrapName:   StringPtr;
End;

pScrapStuff = ^ScrapStuff;
```

Routines

```
function GetScrap (hDest: Handle; what: ResType;
    var offset: LongInt): Longint;

function InfoScrap: pScrapStuff;

function LoadScrap: LongInt;

function PutScrap (length: LongInt; what: ResType; source: Ptr): LongInt;

function UnloadScrap: LongInt;

function ZeroScrap: LongInt;
```

Segment Loader

Constants

const

```
{constants for message returned by the finder on launch}
appOpen  = 0;
appPrint = 1;
```

Types

type

```
{for application parameter}
appFile = record
    vRefNum: Integer;
    ftype:   OSType;
    versNum: Integer; {versNum in high byte}
    fName:   str255;
end; {appFile}
```

Routines

```
procedure UnLoadSeg (routineAddr: Ptr);

procedure ExitToShell;

procedure GetAppParms (var apName: str255; var apRefNum: Integer;
    var apParam: Handle);

procedure CountAppFiles (var message: Integer; var count: Integer);

procedure GetAppFiles (index: Integer; var theFile: AppFile);

procedure ClrAppFiles (index: Integer);
```

Serial Driver

Constants

const

```
{baud rate constants}
baud300    = 380;
baud600    = 189;
baud1200   = 94;
baud1800   = 62;
baud2400   = 46;
baud3600   = 30;
baud4800   = 22;
baud7200   = 14;
baud9600   = 10;
baud19200  = 4;
baud57600  = 0;

{SCC channel configuration word}
{driver reset information masks}
stop10     = 16384;
stop15     = -32768;
stop20     = -16384;
noParity    = 0;
oddParity   = 4096;
evenParity  = 12288;
data5       = 0;
data6       = 2048;
data7       = 1024;
data8       = 3072;

{serial driver error masks}
swOverrunErr = 1;
parityErr    = 16;
hwOverrunErr = 32;
framingErr   = 64;

{refNums from the serial ports}
AinRefNum    = -6;      {serial port A input}
AoutRefNum   = -7;      {serial port A output}
BinRefNum    = -8;      {serial port B input}
BoutRefNum   = -9;      {serial port B output}

{serial Port configuration usage constants for SysParmType Config field }
useFree      = 0;
useATalk     = 1;
useAsync     = 2;

{serial driver message constant}
xOffWasSent  = $80;
```

Types

type

```
SPortSel = (SPortA, SPortB);
```

```
SerShk = packed record      {handshake control fields}
    fXOn: Byte;              {XON flow control enabled flag}
    fCTS: Byte;              {CTS flow control enabled flag}
    xon: Char;               {XOn character}
    xoff: Char;              {XOff character}
    errs: Byte;              {errors mask bits}
    evts: Byte;              {event enable mask bits}
    fInX: Byte;              {Input flow control enabled flag}
    null: Byte;              {unused}
end;
```

```
SerStaRec = packed record
    cumErrs: Byte;           {cumulative errors report}
    XOFFSent: Byte;          {XOff Sent flag}
    rdPend: Byte;            {read pending flag}
    wrPend: Byte;            {write pending flag}
    ctsHold: Byte;           {CTS flow control hold flag}
    XOFFHold: Byte;          {XOff flow control hold flag}
end;
```

Routines

```
function RamSDOpen (whichPort: SPortSel): OSErr;
```

```
procedure RamSDClose (whichPort: SPortSel);
```

```
function SerReset (refNum: Integer; serConfig: Integer): OSErr;
```

```
function SerSetBuf (refNum: Integer; serBPtr: Ptr;
    serBLen: Integer): OSErr;
```

```
function SerHShake (refNum: Integer; flags: SerShk): OSErr;
```

```
function SerSetBrk (refNum: Integer): OSErr;
```

```
function SerClrBrk (refNum: Integer): OSErr;
```

```
function SerGetBuf (refNum: Integer; var count: LongInt): OSErr;
```

```
function SerStatus (refNum: Integer; var serSta: SerStaRec): OSErr;
```

Sound Driver

Constants

const

```
SWmode = -1;  
FTmode = 1;  
FFmode = 0;
```

Types

type

```
{for 4-tone sound generation}
```

```
Wave =    packed array[0..255] of Byte;  
WavePtr = ^Wave;
```

```
FTSoundRec = record  
    duration:      Integer;  
    sound1Rate:    LongInt;  
    sound1Phase:   LongInt;  
    sound2Rate:    LongInt;  
    sound2Phase:   LongInt;  
    sound3Rate:    LongInt;  
    sound3Phase:   LongInt;  
    sound4Rate:    LongInt;  
    sound4Phase:   LongInt;  
    sound1Wave:    WavePtr;  
    sound2Wave:    WavePtr;  
    sound3Wave:    WavePtr;  
    sound4Wave:    WavePtr;  
end;
```

```
FTSndRecPtr = ^FTSoundRec;
```

```
FTSynthRec = record  
    mode:      Integer;  
    sndRec:    FTSndRecPtr;  
end;
```

```
FTSynthPtr = ^FTSynthRec;
```

```
Tone = record  
    count:      Integer;  
    amplitude:  Integer;  
    duration:   Integer;  
end;
```

```

Tones = array[0..5000] of Tone;

SWSynthRec = record
    mode: Integer;
    triplets: Tones;
end;

SWSynthPtr = ^SWSynthRec;

freeWave = packed array[0..30000] of Byte;

FFSynthRec = record
    mode: Integer;
    count: Fixed;
    waveBytes: freeWave;
end;

FFSynthPtr = ^FFSynthRec;

```

Routines

```

procedure SetSoundVol (level: Integer);

procedure GetSoundVol (var level: Integer);

procedure StartSound (synthRec: Ptr; numBytes: LongInt;
    CompletionRtn: ProcPtr);

procedure StopSound;

function SoundDone: Boolean;

```

Standard File Package

Constants

const

```
putDlgID   = -3999;  {SFPutFile dialog template ID}

putSave    = 1;      {save button}
putCancel  = 2;      {cancel button}
putEject   = 5;      {eject button}
putDrive   = 6;      {drive button}
putName    = 7;      {editText item for file name}

getDlgID   = -4000;  {SFGetFile dialog template ID}

getOpen    = 1;      {open button}
getCancel  = 3;      {cancel button}
getEject   = 5;      {eject button}
getDrive   = 6;      {drive button}
getNmList  = 7;      {userItem for file name list}
getScroll  = 8;      {userItem for scroll bar}
```

Types

type

```
SFReply = record
    good: Boolean;           {ignore command if FALSE}
    copy: Boolean;           {not used}
    fType: OSType;          {file type or not used}
    vRefNum: Integer;        {volume reference number}
    version: Integer;        {file's version number}
    fName: string[63];      {file name}
end; {SFReply}

SFTypeList = array[0..3] of OSType;
```


Routines

```
procedure SFPutFile (where: Point; prompt: Str255; origName: Str255;  
                    dlgHook: ProcPtr; var reply: SFReply);  
  
procedure SFPPutFile (where: Point; prompt: Str255; origName: Str255;  
                    dlgHook: ProcPtr; var reply: SFReply;  
                    dlgID: Integer; filterProc: ProcPtr);  
  
procedure SFGetFile (where: Point; prompt: Str255; fileFilter: ProcPtr;  
                    numTypes: Integer; typeList: SFTypeList;  
                    dlgHook: ProcPtr; var reply: SFReply);  
  
procedure SFPGetFile (where: Point; prompt: Str255; fileFilter: ProcPtr;  
                    numTypes: Integer; typeList: SFTypeList;  
                    dlgHook: ProcPtr; var reply: SFReply; dlgID: Integer;  
                    filterProc: ProcPtr);
```

System Error Handler

Routines

```
procedure SysError (errorCode:Integer);
```

TextEdit

Constants

const

```
teJustLeft    = 0;
teJustRight   = -1;
teJustCenter  = 1;
```

Types

type

```
TERec = record
    destRect: Rect;           {Destination rectangle}
    viewRect: Rect;           {view rectangle}
    selRect: Rect;            {Select rectangle}
    lineHeight: Integer;      {Current font lineHeight}
    fontAscent: Integer;      {Current font ascent}
    selPoint: Point;          {Selection point (mouseLoc)}
    selStart: Integer;        {Selection start}
    selEnd: Integer;          {Selection end}
    active: Integer;          {<>0 if active}
    wordBreak: ProcPtr;       {Word break routine}
    clicLoop: ProcPtr;        {Click loop routine}
    clickTime: LongInt;       {Time of first click}
    clickLoc: Integer;         {Char. location of click}
    caretTime: LongInt;       {Time for next caret blink}
    caretState: Integer;       {On/active booleans}
    just: Integer;            {fill style}
    TELength: Integer;        {Length of text below}
    hText: Handle;            {Handle to actual text}
    recalBack: Integer;       {<>0 if recal in background}
    recallLines: Integer;     {Line being recalled}
    clicStuff: Integer;       {click stuff (internal)}
    crOnly: Integer;          {set to -1 if CR line breaks only}
    txFont: Integer;          {Text Font}
    txFace: Style;            {Text Face}
    txMode: Integer;          {Text Mode}
    txSize: Integer;          {Text Size}
    inPort: GrafPtr;          {Grafport}
    highHook: ProcPtr;        {Highlighting hook}
    caretHook: ProcPtr;       {Highlighting hook}
    nLines: Integer;          {Number of lines}
    lineStarts: array [0..16000] of Integer; {Actual line starts}
end;
```

```

TEPtr = ^TRec;
TEHandle = ^TEPtr;

CharsHandle = ^CharsPtr;
CharsPtr = ^Chars;

Chars = packed array[0..32000] of Char;

```

Routines

```

procedure TEActivate    ( h: TEHandle );
procedure TECalText     ( h: TEHandle );
procedure TEClick      ( pt: Point; extend: Boolean; h: TEHandle );
procedure TECopy       ( h: TEHandle );
procedure TECut        ( h: TEHandle );
procedure TEdeActivate ( h: TEHandle );
procedure TEdelate     ( h: TEHandle );
procedure TEdispose    ( h: TEHandle );
procedure TEIdle       ( h: TEHandle );
procedure TEInit;
procedure TEKey        ( key: Char; h: TEHandle );
function  TENew        ( dest, view: Rect ): TEHandle;
procedure TEPaste      ( h: TEHandle );
procedure TEScroll     ( dh, dv: Integer; h: TEHandle );
procedure TEssetSelect ( selStart, selEnd: LongInt; h: TEHandle );
procedure TEssetText   ( inText: Ptr; textLength: LongInt; h: TEHandle );
procedure TEInsert     ( inText: Ptr; textLength: LongInt; h: TEHandle );
procedure TEUpdate     ( rUpdate: Rect; h: TEHandle );
procedure TEssetJust   ( just: Integer; h: TEHandle );
function  TEGetText    ( h: TEHandle ): CharsHandle;
function  TEScrapHandle: Handle;
function  TEGetScrapLen: LongInt;
procedure TEssetScrapLen (length: LongInt);
function  TEFfromScrap: OsErr;
function  TEToscrap: OsErr;
procedure SetWordBreak  (wBrkProc: ProcPtr; hTE: TEHandle);
procedure SetClikLoop   (clikProc: ProcPtr; hTE: TEHandle);

{Box drawing utility}
procedure TextBox ( inText: Ptr; textLength: LongInt; r: Rect;
                    style:Integer );

```

Three-Dimensional Graphics

Constants

const

```
radConst = 3754936; {radConst = 57.29578}
```

Types

type

```
Point3D= record
    x: fixed;
    y: fixed;
    z: fixed;
end;
```

```
Point2D= record
    x: fixed;
    y: fixed;
end;
```

```
XfMatrix = array[0..3,0..3] of fixed;
```

```
Port3DPtr = ^Port3D;
```

```
Port3D    = record
    GrPort:          GrafPtr;
    viewRect:        Rect;
    xLeft,yTop,xRight,yBottom: fixed;
    pen,penPrime,eye: Point3D;
    hSize,vSize:      fixed;
    hCenter,vCenter:  fixed;
    xCotan,yCotan:    fixed;
    ident:            Boolean;
    xForm:            XfMatrix;
end;
```

Variables

var

```
thePort3D: Port3DPtr;
```

Routines

```
procedure InitGrf3D (globalPtr: Ptr);

procedure Open3DPort (port: Port3DPtr);

procedure SetPort3D (port: Port3DPtr);

procedure GetPort3D (var port: Port3DPtr);

procedure MoveTo2D (x,y: fixed);
procedure MoveTo3D (x,y,z: fixed);

procedure LineTo2D (x,y: fixed);
procedure LineTo3D (x,y,z: fixed);

procedure Move2D (dx,dy: fixed);
procedure Move3D (dx,dy,dz: fixed);

procedure Line2D (dx,dy: fixed);
procedure Line3D (dx,dy,dz: fixed);

procedure ViewPort (r: Rect);

procedure LookAt (left,top,right,bottom: fixed);

procedure ViewAngle (angle: fixed);

procedure Identity;

procedure Scale (xFactor,yFactor,zFactor: fixed);

procedure Translate (dx,dy,dz: fixed);

procedure Pitch (xAngle: fixed);

procedure Yaw (yAngle: fixed);

procedure Roll (zAngle: fixed);

procedure Skew (zAngle: fixed);

procedure TransForm (src: Point3D; var dst: Point3D);

function Clip3D (src1,src2: Point3D; var dst1,dst2: Point): Boolean;

procedure SetPt3D (var pt3D: Point3D; x,y,z: fixed);

procedure SetPt2D (var pt2D: Point2D; x,y: fixed);
```

Utilities (Toolbox)

Types

type

```
Int64Bit = record
    hiLong: LongInt;
    loLong: LongInt;
end;

CursPtr = ^Cursor;
CursHandle = ^CursPtr;

PatPtr = ^Pattern;
PatHandle = ^PatPtr;
```

Routines

Note: In Lightspeed Pascal, the following Toolbox calls are always generated inline (rather than via traps) by the compiler: BitAnd, BitOr, BitXor, BitNot, Hiword, Loword. In particular, BitAnd, BitOr or BitXor of two Integer values generates a 16-bit rather than 32-bit instruction. These routines, which are normally part of the Toolbox Utilities, are thus not included here.

```
function BitShift (long: LongInt; count: Integer): LongInt;

function BitTst (bytePtr: Ptr; bitNum: LongInt): Boolean;

procedure BitSet (bytePtr: Ptr; bitNum: LongInt);

procedure BitClr (bytePtr: Ptr; bitNum: LongInt);

procedure LongMul (a,b: LongInt; var dst: Int64Bit);

function FixMul (a,b: Fixed): Fixed;

function FixRatio (numer,denom: Integer): Fixed;

function FixRound (x: Fixed): Integer;

procedure PackBits (var srcPtr,dstPtr: Ptr; srcBytes: Integer);

procedure UnPackBits (var srcPtr,dstPtr: Ptr; dstBytes: Integer);

function SlopeFromAngle (angle: Integer): Fixed;
```

```

function AngleFromSlope (slope: Fixed): Integer;

function DeltaPoint (ptA,ptB: Point): LongInt;

function NewString (theString:Str255): StringHandle;

procedure SetString (theString:StringHandle; strNew: Str255);

function GetString (stringID: Integer): StringHandle;

procedure GetIndString (var theString: str255; strListID: Integer;
                        index: Integer);

function Munger (h: Handle; offset: LongInt; ptr1: Ptr;
                 len1: LongInt; ptr2: Ptr; len2: LongInt): LongInt;

function GetIcon (iconID:Integer): Handle;

procedure PlotIcon (theRect: Rect; theIcon: Handle);

function GetCursor (cursorID: Integer): CursHandle;

function GetPattern (patID: Integer): PatHandle;

function GetPicture (picID: Integer): PicHandle;

procedure GetIndPattern (var thePat: Pattern; patListID: Integer;
                        index: Integer);

procedure ShieldCursor (shieldRect: Rect; offsetPt: Point);

procedure ScreenRes (var scrnHRes, scrnVRes: Integer);

```


Utilities (OS)

Constants

const

```
{for "machine" parameter of Environs}
macXLMachine = 0;
macMachine   = 1;
```

Types

type

```
OSErr = Integer;           { error code }

SysParmType = packed record
    Valid: Byte;            {validation field ($A7)}
    ATalkA: Byte;           {AppleTalk node number hint for port A }
    ATalkB: Byte;           {AppleTalk node number hint for port B }
    Config: Byte;           {ATalk port configuration A =
                             bits 4-7, B = 0-3}
    PortA: Integer;         {SCC port A configuration}
    PortB: Integer;         {SCC port B configuration}
    Alarm: LongInt;         {alarm time}
    Font: Integer;          {default font id}
    KbdPrint: Integer;      {high byte = kbd repeat}
                             {high nibble = thresh in 4/60ths}
                             {low nibble = rates in 2/60ths}
                             {low byte = print stuff}
    VolClick: Integer;      {low 3 bits of high byte = volume control}
                             {high nibble of low byte = double
                             time in 4/60ths}
                             {low nibble of low byte = caret
                             blink time in 4/60ths}
    Misc: Integer;          {EEEC EEEE PSKB FFHH}
                             {E = extra}
                             {P = paranoia level}
                             {S = mouse scaling}
                             {K = key click}
                             {B = boot disk}
                             {F = menu flash}
                             {H = help level}

end; {SysParmType}

SysPPtr = ^SysParmType;

QElemPtr = ^QElem;        {ptr to generic queue element}
```

```

QHdr = record
    QFlags: Integer;      {misc flags}
    QHead: QElemPtr;      {first elem}
    QTail: QElemPtr;      {last elem}
end; {QHdr}

QHdrPtr = ^QHdr;

QTypes = (dummyType, vType, ioQType, drvQType, evType, fsQType);

QElem = record
    case QTypes of
        vType:    (vblQelem: VBLTask);          {vertical blanking}
        ioQType:  (ioQElem: ParamBlockRec);      {I/O parameter block}
        drvQType: (drvQElem: DrvQEL);            {drive}
        evType:   (evQElem: EvQEL);              {event}
        fsQType:  (vcbQElem: VCB);               {volume control block}
    end; {QElem}

DateTimeRec = record
    Year,          {1904,1905,...}
    Month,         {1,...,12 corresponding to Jan,...,Dec}
    Day,           {1,...,31}
    Hour,          {0,...,23}
    Minute,        {0,...,59}
    Second,        {0,...,59}
    DayOfWeek: Integer; {1,...,7 corresponding to Sun,...,Sat}
end; {DateTimeRec}

```

Routines

```

function HandToHand (var theHndl: Handle): OSErr;

function PtrToXHand (srcPtr: Ptr; dstHndl: Handle; size: LongInt): OSErr;

function PtrToHand (srcPtr: Ptr; var dstHndl: Handle;
    size: LongInt): OSErr;

function HandAndHand (hand1,hand2: Handle): OSErr;

function PtrAndHand (ptr1: Ptr; hand2: Handle; size: LongInt): OSErr;

function EqualString (str1,str2: Str255;
    caseSens,diacSens: Boolean): Boolean;

procedure UprString (var theString: Str255; diacSens: Boolean);

procedure SetUpA5;      {MOVE.L A5,-(SP)      ; save old A5 on stack
                        MOVE.L CurrentA5,A5 ; get the real A5}

procedure RestoreA5;    {MOVE.L (A7)+,A5      ; restore A5}

```

```

function GetTrapAddress (trapNum: Integer): LongInt;

procedure SetTrapAddress (trapAddr: LongInt; trapNum: Integer);

function SetDateTime (time: LongInt): OSErr;

function ReadDateTime (var time: LongInt): OSErr;

procedure GetDateTime (var secs: LongInt);

procedure SetTime (d: DateTimeRec);

procedure GetTime (var d: DateTimeRec);

procedure Date2Secs (d: DateTimeRec; var s: LongInt);

procedure Secs2Date (s: LongInt; var d: DateTimeRec);

procedure Delay (numTicks: LongInt; var finalTicks: LongInt);

procedure SysBeep (duration: Integer);

procedure Environs (var rom,machine: Integer);

procedure Restart;

function InitUtil: OSErr;

function GetSysPPtr: SysPPtr;

function WriteParam: OSErr;

procedure Enqueue (qElement: QElemPtr; qHeader: QHdrPtr);

function Dequeue (qElement: QElemPtr; qHeader: QHdrPtr): OSErr;

```

Vertical Retrace Manager

Types

```
type
    VBLTask = record
        qLink:   QElemPtr;    {link to next element}
        qType:   Integer;     {unique ID for validity check}
        vblAddr: ProcPtr;     {address of service routine}
        vblCount: Integer;    {count field for timeout}
        vblPhase: Integer;    {phase to allow synchronization}
    end;
```

Routines

```
function VInstall (VBLTaskPtr: QElemPtr): OSErr;

function VRemove (VBLTaskPtr: QElemPtr): OSErr;

function GetVBLQHdr: QHdrPtr;
```

Window Manager

Constants

const

```
{window messages}
wDraw          = 0;
wHit           = 1;
wCalcRgn      = 2;
wNew           = 3;
wDispose       = 4;
wGrow          = 5;
wDrawGIcon     = 6;

{types of windows}
dialogKind     = 2;
userKind       = 8;

{desk pattern resource ID}
deskPatID      = 16;

{window definition procedure IDs}
documentProc   = 0;
dBoxProc       = 1;
plainDBox      = 2;
altDBoxProc    = 3;
noGrowDocProc  = 4;
rDocProc       = 16;

{FindWindow Result Codes}
inDesk         = 0;
inMenuBar      = 1;
inSysWindow    = 2;
inContent      = 3;
inDrag         = 4;
inGrow         = 5;
inGoAway       = 6;

{defProc hit test codes}
wNoHit         = 0;
wInContent     = 1;
wInDrag        = 2;
wInGrow        = 3;
wInGoAway      = 4;

{axis constraints for DragGrayRgn call}
noConstraint   = 0;
hAxisOnly      = 1;
vAxisOnly      = 2;
```

Types

type

```
WindowPtr    = GrafPtr;
WindowPeek   = ^WindowRecord;

WindowRecord = record
    port:           GrafPort;
    windowKind:     Integer;
    visible:        Boolean;
    hilited:        Boolean;
    goAwayFlag:     Boolean;
    spareFlag:      Boolean;
    strucRgn:       RgnHandle;
    contrRgn:       RgnHandle;
    updateRgn:      RgnHandle;
    windowDefProc:  Handle;
    dataHandle:     Handle;
    titleHandle:    StringHandle;
    titleWidth:     Integer;
    ControlList:    ControlHandle;
    nextWindow:     WindowPeek;
    windowPic:      PicHandle;
    refCon:         LongInt;
end;
```

Routines

```
procedure ClipAbove      (window: WindowPeek);
procedure PaintOne       (window: WindowPeek; clobbered: RgnHandle);
procedure PaintBehind    (startWindow: WindowPeek; clobbered: RgnHandle);
procedure SaveOld        (window: WindowPeek);
procedure DrawNew        (window: WindowPeek; fUpdate: Boolean);
procedure CalcVis        (window: WindowPeek);
procedure CalcVisBehind  (startWindow: WindowPeek; clobbered: RgnHandle);
procedure ShowHide       (window: WindowPtr; showFlag: Boolean);
function  CheckUpdate    (var theEvent: EventRecord): Boolean;
procedure GetWMgrPort    (var wPort: GrafPtr);
procedure InitWindows;
function  NewWindow      (wStorage: Ptr; boundsRect: Rect; title: Str255;
    visible: Boolean; theProc: Integer;
    behind: WindowPtr; goAwayFlag: Boolean;
    refCon: LongInt): WindowPtr;

procedure DisposeWindow  (theWindow: WindowPtr);
procedure CloseWindow    (theWindow: WindowPtr);
procedure MoveWindow     (theWindow: WindowPtr; h,v: Integer;
    BringToFront: Boolean);
procedure SizeWindow     (theWindow: WindowPtr; width,height: Integer;
    fUpdate: Boolean);
```

```

function GrowWindow      (theWindow: windowPtr; startPt: Point;
                           bBox: Rect):LongInt;
procedure DragWindow     (theWindow: WindowPtr; startPt: Point;
                           boundsRect: Rect);
procedure ShowWindow     (theWindow: WindowPtr);
procedure HideWindow     (theWindow: WindowPtr);
procedure SetWTitle      (theWindow: WindowPtr; title: Str255);
procedure GetWTitle      (theWindow: WindowPtr; var title: Str255);
procedure HiliteWindow   (theWindow: WindowPtr; fHiLite: Boolean);
procedure BeginUpdate    (theWindow: WindowPtr);
procedure EndUpdate      (theWindow: WindowPtr);
procedure SetWRefCon     (theWindow: WindowPtr; data: LongInt);
function  GetWRefCon     (theWindow: WindowPtr): LongInt;
procedure SetWindowPic   (theWindow: WindowPtr; thePic: PicHandle);
function  GetWindowPic   (theWindow: WindowPtr): PicHandle;
procedure BringToFront   (theWindow: WindowPtr);
procedure SendBehind     (theWindow,behindWindow: WindowPtr);
function  FrontWindow:   WindowPtr;
procedure SelectWindow   (theWindow: WindowPtr);
function  TrackGoAway    (theWindow: WindowPtr; thePt: Point): Boolean;
procedure DrawGrowIcon   (theWindow: WindowPtr);
procedure ValidRect      (goodRect: Rect);
procedure ValidRgn       (goodRgn: RgnHandle);
procedure InvalRect      (badRect: Rect);
procedure InvalRgn       (badRgn: RgnHandle);
function  FindWindow     (thePoint: Point;
                           var theWindow: WindowPtr): Integer;
function  GetNewWindow   (windowID: Integer; wStorage: Ptr;
                           behind: WindowPtr): WindowPtr;
function  PinRect        (theRect: Rect; thePt: Point): LongInt;
function  DragGrayRgn    (theRgn: RgnHandle; startPt: Point;
                           boundsRect, slopRect: Rect;axis: Integer;
                           actionProc: ProcPtr): LongInt;

```

Appendix F

Error Messages

This Appendix is a guide to the error messages. Compile, link, and the common runtime error messages, with their explanations, are arranged in alphabetical order. There are cross references to other Chapters in this book and examples of incorrect and correct code fragments. The examples come in two classes:

WRONG / RIGHT - an example statement which will cause the error, and a corrected version of it.

ERROR / OK - examples of an erroneous statement and a statement used correctly. Without understanding the programmer's true intentions, it is difficult to provide a corrected version.

Error messages are displayed in an Alert box, unless an error occurs while evaluating an expression in the Observe Window. In that case, an abbreviated error message is displayed in the left cell of the Observe Window.

The examples have also taken advantage of Lightspeed Pascal's relaxation of order and number of declarations in a Block. To make this Appendix concise, the following declarations are assumed to be made:

```
const
  aConstant = 42;
  aCharConstant = 'c';
  aStringConstant = 'hi there';
type
  ColorType = (red, orange, yellow, green, blue, violet);
  WeightType = (LIGHT, MEDIUM, HEAVY);
  aType = integer;
var
  aBool : boolean;
  aChar : char;
  anInt : integer;
  aLong : longint;
  aReal : real;
  aDouble : double;
  anExt : extended;
  aString : string;
  aColor : ColorType;
  aPtr : Ptr;          { Ptr is a predefined type that points to -128..127 }
  ColorSet : set of ColorType;
  TextFile : Text;
  FileOfInt : file of integer;
```


"symbol" is already declared at this level.

See 2.2.4 of the Language Reference. Example:

```
var
  aDupName : integer;
  aDupName : real;           { Error: aDupName already declared }
type
  recordType = record
    aDupName : integer;      { OK: not yet declared within the record }
    aDupName : real;         { Error: aDupName already declared in record }
  end;
procedure aDupName;          { Error: aDupName already declared as a variable }
begin
end;
procedure P1 (aDupName:char); { OK: aDupName is not declared at this level }
begin
end;
procedure P2 (X:char;
              X:real);        { Error: X declared twice in parameter list }
begin
end;
```

"symbol" is not declared.

- Check your spelling. Be careful to distinguish between 1 (one), l (lower case L), and I (upper case i); and between 0 (zero) and O (oh). (Pascal is not case sensitive.)
- Make sure the symbol is declared before it is used.
- Make sure the symbol is visible at the level you are trying to use it.
- If the symbol is in another unit, don't forget to use the unit name in a **uses** clause.

In almost all cases, a name must be defined (or predefined) before it can be used. See Sections 3, 4, and 7 of the Language Reference. Example:

```
program HasUndeclaredSymbols;
type
  BadType = TypeYetToBeDefined; { Error }
  TypeYetToBeDefined = integer; { This must come before the previous statement }
  RecPtrType = ^RecType;        { OK: a pointer type to a type to be defined later }
  RecType = record
    PtrToARecord : ^UndefinedType; { Error }
    PtrToNextRecord : ^RecType;     { Wrong: RecType declaration isn't finished }
    PtrToNextRecord : RecPtrType;   { Right: use RecPtrType instead }
    aField : integer;
  end;
var
  r : RecordType;
procedure Proc;
var
  PrivateToProc : integer;
begin
end;
```

```

begin
  PrivateToProc := 4;           { Error: PrivateToProc isn't declared at this level }
  aField := 4;                 { Wrong }
  r.aField := 4;               { Right }

  UndeclaredVariable := 4;     { Error }
  UndeclaredProcedure;        { Error }
  r.aField := UndeclaredFunction; { Error }
end.

```

"symbol" isn't in the current project, hasn't been successfully compiled, or is in the wrong build order.

Before a file that uses Units can be compiled two things must be true: (1) the Units used must be successfully compiled into the Project and (2) in the build order, the Units being used must precede the files that use them. See Chapter 5 and Section 8.5 of the Language Reference.

Note for Macintosh Pascal users: QUICKDRAW1 and QUICKDRAW2 are now predefined libraries, and therefore, are no longer needed or accepted in a uses-clause. If you use other Standard Macintosh Libraries (e.g. SANE), see Chapter 8.

Example:

```

unit anUnit;
interface
  uses
    NonExistantUnit, UncompiledUnit, UnitFollowingThisOne;    { Error }
  implementation
end.

```

"symbol" looks like it's being used as a function, but it isn't a function name.

See 5.2 of the Language Reference. Example:

```

aLong := anInt(8);    { Error }

```

"symbol" looks like it's being used as a procedure, but it isn't a procedure name.

See 6.1.2 of the Language Reference. Example:

```

type
  IntPtrType = ^integer;
var
  anIntPtr : IntPtrType;
function Func : char;
begin
end;

```

```

begin
  aConstant;      { Error }
  aType;          { Error }
  aChar;          { Error }

  Func;           { Wrong: the result returned by a function must be used }
  aChar := Func;  { Right }

  IntPtrType(aPtr) := 3;      { Wrong: can't cast Lvalues as you can in LisaPascal }

  anIntPtr := pointer(aPtr); { pointer is in 10.2.6 of the Language Reference }
  anIntPtr^ := 3;           { Right: use temporary variable anIntPtr }
end;

```

"symbol" was previously declared as a function, not a procedure.

When a function is declared in an **interface** part or as a *forward* function, it cannot later be defined as a procedure. See 7.1.1 and 8.1 of the Language Reference. Example:

```

unit MisdefinedFunctions;
interface
  function InterfaceFunc : char; { To be defined in IMPLEMENTATION part }
implementation
  procedure InterfaceFunc;        { Wrong: FUNCTION declaration expected }
  begin
  end;
  function InterfaceFunc;        { Right }
  begin
  end;

  function ForwardFunc : char;   { To be defined below }
  forward;
  procedure ForwardFunc;         { Wrong: FUNCTION declaration expected }
  begin
  end;
  function ForwardFunc;         { Right }
  begin
  end;
end.

```

"symbol" was previously declared as a procedure, not a function.

When a procedure is declared in an **interface** part or as a *forward* procedure, it cannot later be defined as a function. See 7.1.1 and 8.1 of the Language Reference. Example:

```

unit MisdefinedProcedures;
interface
  procedure InterfaceProc;       { To Be defined in IMPLEMENTATION part }
implementation
  function InterfaceProc : char; { Wrong: PROCEDURE definition expected }
  begin
  end;
  procedure InterfaceProc;      { Right }
  begin
  end;
end;

```

```

procedure ForwardProc;           { To Be defined below }
forward;
function ForwardProc : char;     { Wrong: PROCEDURE definition expected }
begin
end;
procedure ForwardProc;           { Right }
begin
end;
end.

```

@ can only be applied to variable references, or to top-level procedure or function names.

The address operator @ can only be applied to objects that actually have memory allocated to them. For example, constants, types, etc., do not allocate memory. Furthermore, @ cannot be applied to sub-level subroutines, because references to the variables in the outer scope cannot be set up properly. See 5.1.6 and 5.1.6.4 of the Language Reference. Example:

```

program AtSigns;
procedure Proc;
  procedure SubProc;
  begin
    aPtr := @p;           { OK }
    aPtr := @SubProc;     { Error: SubProc is not a top level procedure }
  end;
begin { Proc }
end;

begin
  aPtr := @anInt;         { OK: memory is allocated for anInt }
  aPtr := @Proc;         { OK }
  aPtr := @aConstant;    { Error: aConstant is a constant }
  aPtr := @orange;       { Error: orange is an enumerated constant }
  aPtr := @ColorType;    { Error: ColorType is a type }
  aPtr := @integer;      { Error: integer is a type }
end.

```

@ can't be applied to a component of a packed type.

@ cannot be applied to components of packed structures, because the actual data representation of a component in a packed structure may be different from that actual data representation in an unpacked structure. However, @ can be applied to the packed structure as a whole. See 5.1.6.2 of the Language Reference. Example:

```

var
  aPACKEDRecord : packed record
    field : integer;
  end;
  aRecord : record
    field : integer;
  end;
  aPACKEDArray : packed array[1..4] of char;
  anArray : array[1..4] of char;

```

```

begin
  aPtr := @aPACKEDRecord.field; { Wrong }
  aPtr := @aRecord.field;      { Right: record is not packed }
  aPtr := @aPACKEDRecord;      { OK: not taking address of component field }

  aPtr := @aPACKEDArray[2];    { Wrong }
  aPtr := @anArray[2];        { Right: array is not packed }
  aPtr := @aPACKEDArray;      { OK: not taking address of component element }
end;

```

@ can't be applied to predefined or inline routines.

The @ operator can only be applied to actual routines, not predefined, inline, or Macintosh Toolbox routines (which don't generate real subroutine calls). However, you can write a glue routine which wraps itself around the desired routine. Then, the @ operator can be applied to that glue routine. See 5.1.6.4 of the Language Reference. Example:

```

procedure NOP;
inline
  $4e71; { M68000 NOP instruction }
procedure WritelnGlue (s : string);
begin
  writeln(s);
end;
procedure NOPGlue;
begin
  NOP;
end;
function ButtonGlue : boolean;
begin
  ButtonGlue := Button;
end;
begin
  aPtr := @writeln;      { Wrong: writeln is predefined procedure }
  aPtr := @writelnGlue;  { Right: writeln is in a glue routine }
  aPtr := @NOP;          { Wrong: NOP is an inline procedure }
  aPtr := @NOPGlue;      { Right: NOP is in a glue routine }
  aPtr := @Button;       { Wrong: Button is an Toolbox routine }
  aPtr := @ButtonGlue;   { Right: Button is in a glue routine }
end.

```

Allocation failed in NEW.

This runtime error most frequently occurs when too much memory is allocated by a memory allocation call (e.g. *new*). This error could also occur if the heap is corrupted. Increasing the Heap Size with the **Run Options...** command may solve your problem. See 10.1.1 of the Language Reference and Chapter 6. Example:

```

type
  RecordType = record
    a: array[1..256] of longint;
  end;
var
  RecordPtr : ^RecordType;

```

```

begin
  while true do
    new(RecordPtr);    { Eventually the heap will be exhausted }
  end;

```

Array index type incompatibility.

The type of the expression that indexes an array must be compatible with the type of the index in the array declaration. See 4.3.1 of the Language Reference. Example:

```

var
  ai : array[1..10] of char;
  aC : array[ColorType] of char;
begin
  ai[4]:= 'a';           { OK: 4 is within 1..10 }
  ai[4 + anInt]:= 'a';   { OK: type of (4+anInt) is compatible with 1..10 }

  ai[aReal]:= 'x';       { Wrong }
  ai[round(aReal)]:= 'x'; { Right }

  ai[aColor]:= 'x';      { Wrong }
  ai[ord(aColor)]:= 'x'; { Right: but why bypass strong typing? }
  aC[aColor]:= 'x';      { Right: Best }

  ai['c']:= 'x';        { Error }
  ai[4 + aReal]:= 'x';   { Error: type of (4+aReal) is Real, not 1..10 }
  ai[succ(aColor)]:= 'x'; { Error: type of succ(aColor) is ColorType, not 1..10 }
  ai[ai[anInt]]:= 'x';   { Error: type of ai[anInt] is char, not 1..10 }

  aChar:= aString[aChar]; { Wrong: string index must be 1..255 }
  aChar:= aString[ord(aChar)]; { Right }
end;

```

Array index type is not integer, char, enumerated, or subrange.

See 3.2.1 of the Language Reference. Example:

```

var
  Good_1 : array['A'..'Z'] of char;    { OK: char subrange }
  Good_2 : array[ColorType] of char;    { OK: enumerated type }
  Good_3 : array[1..10] of char;        { OK: integer subrange }

  Bad_1 : array[real] of char;          { Error }
  Bad_2 : array[40000..40001] of char;  { Error: longint subrange is bad }

```

Assignment type incompatibility.

You can't put a square peg in a round hole. Pascal catches illogical statements that try to assign an expression of one type into a variable of an assignment-incompatible type. On occasion, this rule needs to be broken. The following examples show common errors and methods to break the rules. Use the methods at your own risk. Assignment compatibility can be subtle. See 3.5.3 and 5.4 of the Language Reference. Example:

```
type
  CanonicalType = array[1..4] of char; { a type }
  IdenticalType = CanonicalType;      { an identical type with CanonicalType }
  AnotherType = array[1..4] of char;  { a type DIFFERENT from CanonicalType }
var
  AnonArray : array[1..4] of char;    { an array of an anonymous type }
  CanonicalArray : CanonicalType;     { a variable of a named type }
  IdenticalArray : IdenticalType;     { a variable of a same named type }
  AnotherArray : AnotherType;         { a variable of a different named type }

  AC : array[1..4] of char;
  PAC : packed array[1..4] of char;
  Rec : packed record
    i : integer;
  end;
  pInt : ^integer;
  pChar : ^char;
begin
  aString:=AC;      { Wrong: can't assign UNpacked array of char to strings }
  aString:=PAC;     { Right: OK to assign pack array of char to strings }
  PAC:=aString;     { Right: OK to assign strings to pack array of char }
  AC:=PAC;          { Error: can't assign packed to unpacked array }

  anInt:=aReal;     { Wrong: Pascal doesn't do float to integer conversions }
  anInt:=round(aReal); { Right: explicit integer to float conversion }
  anInt:=trunc(aReal); { Right }

  aReal:=anInt;     { OK: Pascal does do integer to float conversions }
  aReal:=3;         { OK }

  aColor:=green;    { OK }
  i:=green;         { Wrong: can't assign enumerated type to integer }
  i:=ord(green);    { Right: but why bypass strong typing? }
  aColor:=i;        { Wrong: can't assign integer to enumerated }
  aColor:=ColorType(i); { Right: a CAST works but why bypass strong typing? }

  AC:=Rec;          { Error: can't assign totally different types }

  pInt:=pChar;      { Wrong: can't assign different pointer types }
  pInt:=pointer(pChar); { Right: a CAST works but why bypass strong typing? }

  pInt := 7;        { Wrong: can't assign a pointer a numerical value }
  pInt^ := 7;       { Right }

  CanonicalArray := IdenticalArray; { OK }
  CanonicalArray := AnotherArray;   { Error: the types aren't assignment compatible }
  CanonicalArray := AnonArray;      { Error: anonymous types are never assignment... }
                                     { ...compatible. }
end;
```

At A-Trap

This runtime message is displayed in the Observe Window when the **Break At A-Traps** command is checked, and the expression being observed, directly or indirectly, makes a Macintosh Toolbox/Operating System call. See Chapter 12.

At least one comment of more than 255 characters has been truncated to 255.

Lightspeed Pascal only supports comment lines which have fewer than 256 characters.

At least one identifier, literal string, or other token of more than 255 characters has been truncated to 255.

Lightspeed Pascal only supports identifiers, string literals, and other tokens which have fewer than 256 characters.

At least one valid constant declaration must follow CONST.

See 1.7 and 2.1 of the Language Reference. Example:

```
const
  BadConstant : false;    { Wrong: Colon(:) instead of Equal(=) }
  GoodConstant = true;    { Right }

const
                                { Error: must have at least one CONST declaration }

begin
end;
```

At least one valid type declaration must follow TYPE.

See 2.1 and 3 of the Language Reference. Example:

```
type
  BadType : integer;      { Wrong: Colon(:) instead of Equal(=) }
  GoodType = integer;     { Right }

type
                                { Error: must have at least one TYPE declaration }

begin
end;
```


At least one valid variable declaration must follow VAR.

See 2.1 and 4.1 of the Language Reference.

```
var
  BadVariable = integer;      { Wrong: Equal(=) instead of Colon(:) }
  GoodVariable : integer;     { Right }
var
                                { Error: must have at least one VAR declaration }
begin
end;
```

Autointerrupt Exception Detected

This runtime message is given when the programmer's switch is hit, but Macsbug is not installed. This should only be used as a last resort. Since you cannot be sure of the state of your program, it may crash if you restart or reset it. Therefore, save all your files before continuing. It's better to run with code compiled with the Debug Option (see Chapter 13) and click on the Bug Spray Can to stop, or always have Macsbug installed. See Appendix G.

Available memory for variables declared at this level has been exhausted.

Because of the MC 68000 architecture, the local storage per function or procedure must not exceed 32766 bytes. Similarly, the total global storage must not exceed 32766 bytes. Excessive global storage may not be detected until link time.

In practice, even less storage may be available. Subroutines require local storage for all temporary variables generated by the compiler. For applications, QuickDraw globals are included in global storage. Furthermore, Libraries may also require some global storage.

If you need more memory, you must use storage allocation subroutines. See 10.1 and 10.2 of the Language Reference. For large data structures, the use of Handles is highly recommended. See *Inside Macintosh*.

```
program MemoryHog;
const
  HalfTooBig = 16500;  { Note the Max global size is approximately 32000 bytes }
  TooBig = 33000;
type
  BigArrayType = packed array[0..HalfTooBig] of char;
var
  BigGlobalArray1 : BigArrayType;      { OK }
  BigGlobalArray2 : BigArrayType;      { Error : Too much memory }
                                         { Better: }
  BigArrayPtr1 : ^BigArrayType;        { Allocate Pointers instead }
  BigArrayPtr2 : ^BigArrayType;
procedure aProc;
var
  BigLocalArray1 : packed array[0..HalfTooBig] of char;  { OK }
  BigLocalArray2 : packed array[0..HalfTooBig] of char;  { Error }
begin
end;
```

```

begin
  New(BigArrayPtr1);           { Allocate memory }
  New(BigArrayPtr2);
  BigArrayPtr1^[15000] := 'c'; { Can access large array via pointers }
end.

```

Bad actual parameter for formal VAR, procedural, or functional parameter.

See sections 7.3.2, 7.3.3, and 7.3.4 of the Language Reference. Example:

```

procedure VarProc (var i : integer);
begin
end;

procedure Proc (i : integer);
begin
end;
procedure CallP ( procedure P (i : integer));
begin
end;
function Func (c : char) : char;
begin
end;
procedure CallF ( function F (c : char) : char);
begin
end;

begin
  VarProc(anInt + 4); { Error: expressions can't be passed as VAR parameters }
  VarProc(anInt);    { OK }

  CallP(Proc(3));    { Wrong: procedural parameters aren't called with parameters }
  CallP(Proc);       { Right }

  CallF(Func('c')); { Wrong: functional parameters aren't called with parameters }
  CallF(Func);       { Right }
end;

```

Bad boolean value.

During runtime, the procedure *read*, *readln*, or *readstring* expected a boolean value (*True* or *False*), but did not get one. See 9.4.1.5 of the Language Reference. Example:

```

begin
  readstring('FaLSe', aBool); { OK: casing not important }

  readstring('NotTrue', aBool); { Wrong }
  readstring('FALSE', aBool);   { Right }

  readstring('NotFalse', aBool); { Wrong }
  readstring('true', aBool);     { Right }
end;

```

Bad decimal-places expression in WRITE, WRITELN, or STRINGOF call.

The decimal-places expression is only valid for real output expressions and must evaluate to a positive integer expression. See 9.4.3.1, 9.4.3.2, 9.4.3.3, 9.4.3.4, 9.4.3.5, 9.4.3.6, and 9.4.3.7 of the Language Reference. Example:

```
writeln(anInt : 7 : 3);           { Error }
writeln(aString : 7 : 3);        { Error }
writeln(aReal : 7 : aString);    { Error }

aString := stringof(anInt : 7 : 3); { Error }
aString := stringof(aString : 7 : 3); { Error }
aString := stringof(aReal : 7 : aString); { Error }
```

Bad device type for operation.

Some subroutine calls are inappropriate for the device specified. See 9.5 of the Language Reference. Example:

```
close(Output);           { Must first close the predefined file Output }
RESET(Output, 'TextWindow:'); { Error: TextWindow is a writeonly device, so }
                           { RESET won't work }
OPEN(Output, 'TextWindow:'); { Error: TextWindow is a writeonly device, so }
                           { OPEN won't work }
rewrite(Output, 'TextWindow:'); { OK: can only open TextWindow for writeonly access }
RESET(f, 'PRINTER:');      { Error: Printer is a writeonly device, so }
                           { RESET won't work }
OPEN(f, 'PRINTER:');       { Error: Printer is a writeonly device, so }
                           { OPEN won't work }
rewrite(f, 'PRINTER:');    { OK: can only open Printer for writeonly access }
```

Bad enumerated value.

During runtime, the procedure *read*, *readln*, or *readstring* expected an enumerated value for a specific type, but did not get one. See 9.4.1.5 of the Language Reference. Example:

```
readstring('paisley', aColor); { Error }
readstring('plaid', aColor);   { Error }

readstring('violet', aColor);  { OK }
readstring('yeLLow', aColor);  { OK: casing not important }
```

Bad expression type for READSTRING, or for READ or READLN from a text file.

Text files and strings can only contain character oriented data. Values which have a string representation (except for Packed arrays of characters) can be read from a Text file or extracted from a string. For example, an integer can be represented by a string of digits and can be read from a Text file. On the other hand, a Record cannot be read from a Text file, because it is a collection of data which does not have a simple string representation. You may think it should be represented in a particular string format, but there are no rules for it in Pascal.

Values that cannot be read from a Text file, can be read from a **file of** that value's type. For example, an entire Record can be read from a **file of** that Record's type. If you really want to read a Record from a Text file or string, then read each field of the Record from the file or string. See 9.4.1, 9.4.2, and 10.7.6 of the Language Reference. Example:

```
type
  RecType = record
    Field_1 : integer;
    Field_2 : char;
  end;
var
  InString : string; { Input String that readstring reads from }
  TextFile : text;   { Input that readln reads from }
  FileOfREC : file of RecType;

  IntPtr : ^integer;
  anArray : array[1..2] of char;
  PAC : packed array[1..6] of char;
  Rec : RecType;
begin
  open(TextFile, 'text file');
  open(FileOfREC, 'file of records');
  readln(TextFile, aChar, aString, aColor, anInt, aLong, aReal); {OK}

  readln(TextFile, PAC);           { Error: readln can't convert string to PAC }

  readln(TextFile, IntPtr);        { Wrong }
  readln(TextFile, aLong);         { OK, but... }
  IntPtr := pointer(aLong);        { ...probably not what you really want to do }
  readln(TextFile, ^IntPtr);       { Right }

  readln(TextFile, anArray);        { Wrong }
  readln(TextFile, anArray[1], anArray[2] ); { Right }

  readln(TextFile, Rec);            { Wrong: can't read a record from a Text file }
  readLN(FileOfREC, Rec);           { Wrong: can't input a record with a NEWLINE }
  read(FileOfREC, Rec);             { Right: OK to read a record from a file of records }
  readln(TextFile, Rec.Field_1, Rec.Field_2); { Right: OK to read fields from TextFile }

  readstring(InString, aChar, aString, aColor, anInt, aLong, aReal); {OK}
  readstring(InString, anArray);   { Error }
  readstring(InString, Rec);       { Error }
end;
```

Bad expression type for STRINGOF, or for WRITE or WRITELN to a text file.

Text files can only contain character oriented data. Values which have a string representation can be written to a Text file or copied to a string. For example, an integer can be represented by a string of digits and can be written to a Text file. On the other hand, a Record cannot be read from a Text file because it is a collection of data that does not have a simple string representation. You may think it should be represented in a particular string format, but there are no rules for it in Pascal.

Values that cannot be written out to a Text file can be written out to a **file of** that value's type. For example, a Record can be written to a **file of** that Record's type. If you really want to write a Record to a Text file, then write each field of the Record to the Text file or copy each field to a string. See 9.4.3, 9.4.4, and 10.7.5 of the Language Reference. Example:

```
type
  RecType = record
    Field_1 : integer;
    Field_2 : char;
  end;
var
  IntPtr : ^integer;
  anArray : array[1..2] of char;
  PAC : packed array[1..6] of char;
  aRec : RecType;
  FileOfREC : file of RecType;
  Result : string;
begin
  open(TextFile, 'text file');
  open(FileOfREC, 'file of records');

  writeln(TextFile, aBool, aChar, aString, aColor, anInt, aLong, aReal, PAC);      {OK}
  Result := stringof(aBool, aChar, aString, aColor, anInt, aLong, aReal, PAC);    {OK}

  writeln(TextFile, IntPtr);              { Wrong }
  writeln(TextFile, ord4(IntPtr));         { OK, probably not what you really want to do }
  writeln(TextFile, IntPtr^);             { Right: outputs what is pointed to by IntPtr }

  writeln(TextFile, anArray);              { Wrong }
  writeln(TextFile, anArray[1], anArray[2]); { Right }

  write(TextFile, aRec);                   { Wrong: can't write a record to a Text file }
  writeln(FileOfREC, aRec);                { Wrong: can't output a record with a NEWLINE }
  write(FileOfREC, aRec);                  { Right: OK to write a record to a file of records }
  write(TextFile, aRec.Field_1, aRec.Field_2 ); { Also Right }

  writeln(ColorSet);                      { Wrong: can't write out a set directly }
  for aColor := red to violet do          { Right: find elements in set and print them }
    if aColor in ColorSet then
      write(aColor);

  Result := stringof(aPtr);                { Error: can't do this with stringof }
  Result := stringof(anArray);             { Error }
  Result := stringof(TextFile);            { Error }
  Result := stringof(aRec);                { Error }
end;
```

Bad field-width expression in WRITE, WRITELN, or STRINGOF call.

The field-width expression must evaluate to an integer expression greater than 0. Also, it is forbidden to use field widths when writing to non-Text files. See 9.4.3.1, 9.4.3.2, 9.4.3.3, 9.4.3.4, 9.4.3.5, 9.4.3.6, and 9.4.3.7 of the Language Reference. Example:

```
anInt := 7;
writeln(anInt : 1 + anInt); { OK }
writeln(anInt : aString);   { Error: field width expression must be an integer }
writeln(anInt : aReal);     { Error: field width expression must be an integer }

aString := stringof(anInt : aString); { Error }
aString := stringof(anInt : aReal);   { Error }

write(FileOfInt, anInt : 4); { Wrong: can't specify field width for Non-Text files }
write(FileOfInt, anInt);     { Right }
```

Bad filename (pathname too long).

With the new Hierarchical File System (HFS), it is possible to have files in several nested folders, which can result in path names having more than 255 characters. Unfortunately, the Macintosh file system does not allow path names to be longer than 255 characters.

Bad pointer in DISPOSE.

This runtime error most frequently occurs when trying to dispose a nil pointer or dispose the same memory twice. A less likely cause is that the heap is corrupted. This can happen if memory is used after it has been disposed. Use pointers with great caution! See 10.1.2 of the Language Reference.

Bad set constructor.

See 5.3 of the Language Reference.

```
ColorSet := [[]]; { Wrong: can't have a set of an empty set }
ColorSet := [];   { Right: Empty Set }
```

Bad SIZEOF parameter.

The predefined function *sizeof* can only take variable or type names. (Note: the size of a string is not the same as its length, see 3.3 and 4.3.1 of the Language Reference). *sizeof* is described in 10.7.1 of the Language Reference. Example:

```
type
  LongType = longint;    { sizeof = 4 }
var
  size : integer;
  string19 : string[19];
  string20 : string[20];
  pr2 : packed record { sizeof = 2 : packed chars take up 1 byte apiece }
    c1, c2 : char;
  end;
  pr3 : packed record { sizeof = 4 : extra byte to align even word }
    c1, c2, c3 : char;
  end;
  ur2 : record          { sizeof = 4 : unpacked chars take up 2 bytes apiece }
    c1, c2 : char;
  end;
  array10 : array[1..10] of integer;
procedure aProcedure;
begin
end;
function aFunction : boolean;
begin
end;
begin
  size := sizeof(anInt + aLong); { Error: sizeof doesn't work with expressions }
  size := sizeof(aProcedure);    { Error: sizeof doesn't work with procedures }
  size := sizeof(aFunction);     { Error: sizeof doesn't work with functions }
  size := sizeof(aConstant);     { Error: sizeof doesn't work with constants }
  size := sizeof(pr2.c1);        { Error: sizeof doesn't work with record components }
  size := sizeof(array10[1]);    { Error: sizeof doesn't work with array elements }

  size := sizeof(LongType);      { OK: size = 4 }
  size := sizeof(string19);      { OK: size = 20 = 1 bytecount + 19 databytes }
  size := sizeof(string20);      { OK: size = 22 = 1 bytecount + 20 databytes + 1 padbyte }
  size := sizeof(pr2);           { OK: size = 2, packed chars take up 1 byte apiece }
  size := sizeof(pr3);           { OK: size = 4, pad byte to align even word }
  size := sizeof(ur2);           { OK: size = 4, unpacked chars take up 2 bytes apiece }
  size := sizeof(array10);       { OK: size = 20, 10 elements of 2 bytes = 20 }

  size := sizeof(size);          { OK: size = 2 }
  size := sizeof(boolean);       { OK: size = 1 }
  size := sizeof(char);          { OK: size = 2, NOTE: unpacked chars take up two bytes }
  size := sizeof(integer);       { OK: size = 2 }
  size := sizeof(longint);       { OK: size = 4 }
  size := sizeof(real);          { OK: size = 4 }
  size := sizeof(double);        { OK: size = 8 }
  size := sizeof(extended);      { OK: size = 10 }
end;
```

Boolean expression required.

Boolean expressions are required by control statements and boolean operators. See 6.2.2.1, 6.2.3.1, 6.2.3.2, 5, and 5.1.3 of the Language Reference. Example:

```
aBool := 3 or 4;    { Error: the numbers 3 and 4 aren't booleans }
if 3 + 4 then      { Error: (3+4) doesn't evaluate to True or False }
  repeat
    until aReal;    { Error: aReal doesn't evaluate to True or False }
  while green do    { Error: green doesn't evaluate to True or False }

aBool := lower < anInt and anInt < upper;
                { Wrong: AND has a higher precedence then =, <>, etc. }
aBool := (lower < anInt) and (anInt < upper);
                { Right }
```

Can't find the file "*filename*". Would you like to look for it?

Lightspeed Pascal remembers where a Project's files are by volume name (and pathname if you are using the Hierarchical File System). When this dialog box appears, the file is no longer where the Project thinks it is. Either you have deleted the file or moved it to a new location. If you click OK to this dialog and find the file, the Project will remember its new location and ask you:

Change "*old location*" to "*new location*" in all SUBSEQUENT (by build order) Project entries also?

Lightspeed Pascal thinks if you have moved one file, then it is likely that you have moved other files too. As a convenience, you can change the references from the old location to the new location, but only by build order starting with files after the one you just changed.

Can't read Keyboard or Modem.

This runtime error occurs when an expression in the Observe Window tries to get input from the keyboard or modem. For example, the value cell of the Observe Window will display this error when evaluating either of the following functions:

```
var
  s : string;
  f : text;
function ReadKeyBoard : integer;
begin
  readln(s);
  ReadMe := length(s);
end;
function ReadModem : integer;
begin
  reset(f, 'MODEM:');
  readln(f, s);
  ReadMe := length(s);
  close(f);
end;
```


Case constant incompatible with tag-type or selector expression.

See 3.2.2 and 6.2.2.2 of the Language Reference. Example:

```
type
  BadVariantRecordType = record
    case Tag : ColorType of
      red : (          { OK: red is a ColorType }
        i : integer;
      );
      15 : (           { Error: 15 is not a ColorType }
        j : real;
      );
    end;
begin
  case aColor of
    red :              { OK: red is a ColorType }
      writeln('OK');
    15 :               { Error: 15 is not a ColorType }
      writeln('Error');
  end
end;
```

Case constant needed here.

See 6.2.2.2 of the Language Reference. Example:

```
var
  aVariantRecord : record
    case boolean of
      : ( { Wrong: missing case constant before the colon (:) }
        i : integer
      );
      true : ( { Right: true is a valid case constant }
        c : char
      );
    end;
begin
  case anInt of
    :
      writeln('Wrong: missing case constant before colon (:)');
    2 :
      writeln('Right: 2 is a valid case constant');
    3..6 :
      writeln('Wrong: Subranges in Case List is not allowed in Lightspeed Pascal');
    3, 4, 5, 6 :
      writeln('Right: 3,4,5,6 are valid case constants');
  end; { of case }

  case anInt of
    { Error: at least one case constant must be in a case statement }
  end; { of empty case }
end;
```

Case selector out of range. (Trap 4)

See 6.2.2.2 of the Language Reference. Example:

```
anInt := 3;
case anInt of
1:
;
end;           { Error: 3 isn't a tag in this CASE. Should use OTHERWISE }
```

Change "*old location*" to "*new location*" in all SUBSEQUENT (by build order) Project entries also?"

See the Error Message:

"Can't find the file '*filename*'. Would you like to look for it?"

CLOSE of an unnamed file not allowed.

Unnamed files are sometime also referred to as "anonymous" files. See 9.1 and 9.2.4 of the Language Reference. Example:

```
var
f : Text;
begin
rewrite(f); { Associate an unnamed file with f }
close(f);   { Wrong: can't close an unnamed file }
end;        { When the scope of the file variable is exited, f will be closed }
```

Colon (:) required on this line or above.

Colons are needed in labeled statements and case statements. See 3.2.2, 6.1.3, and 6.2.2.2 of the Language Reference. Example:

```
procedure Proc;
label
1, 2;
begin
goto 1;
1           { Wrong: missing colon after the label 1 }
writeln;
goto 2;
2 :         { Right }
writeln;

case anInt of
12+1 :
writeln('Wrong: case tags cannot be constant expressions');
13 :
writeln('Right: 13 is a constant');
end;
end;
```

Component type of a file can't contain a file type.

See 3.2.4 of the Language Reference. Example:

```
type
  FileType = file of integer;
  Bad_NestedFileType = file of FileType;    { Error: FILE OF a file type }
  Bad_NestedFileRecordType = file of record
    F : FileType;                          { Error: FILE OF a type containing file }
  end;
  Bad_FileOfText = file of Text;           { Error: Text is a predefined file type }
```

Constant expected. "symbol" isn't a constant.

At the erroneous statement, the compiler needs a constant to generate code for storage allocation, case statements, or variant records. See section 3 and 6.2.2.2 of the Language Reference. Example:

```
var
  NotAConstant : integer;

  aBadString : string[NotAConstant];      { Wrong }
  aGoodString : string[aConstant];       { Right }

  Bad : array[1..NotAConstant] of char;  { Wrong }
  Good : array[1..aConstant] of char;    { Right }
begin
  case anInt of
    NotAConstant :                      { Wrong: case tags must be constants }
      writeln('Equal to NotAConstant');
    aConstant :                          { Right: this case tags is constants }
      writeln('Equal to aConstant');
    otherwise
      writeln('Otherwise something else')
  end;

  { If you really want to have variable case tags, use a series of else-ifs }
  if anInt = NotAConstant then          { Right }
    writeln('Equal to NotAConstant')
  else if anInt = aConstant then        { Right }
    writeln('Equal to aConstant')
  else
    writeln('Otherwise something else') { Right }
end;
```

Constant, expression, or packed type component was passed to a formal VAR parameter.

When a parameter is passed to a formal **var** parameter, the called subroutine can change the actual parameter passed, not just a copy of it. Therefore, constants and expressions cannot be passed as **var** parameters, because they cannot be changed.

Components of packed structures cannot be passed as formal **var** parameters, because the actual data representation of a component in a packed structure may be different from the actual data representation in an unpacked structure. (For this Pascal implementation, only the representation of packed and unpacked chars are different.) You can pass the entire packed structure as a **var** parameter. Note: the predefined procedures *read*, *readln*, and *readstring* are special because they can be passed components of packed records. See 7.3.2 of the Language Reference. Example:

```
type
  PackedArrayIntType = packed array[1..4] of integer;

var
  PackedRec : packed record
    i1, i2 : integer;
  end;

  Rec : record
    i1, i2 : integer;
  end;

  PackedAI : PackedArrayIntType;
  AI : array[1..4] of integer;

procedure NextInt (var i : integer);
begin
  i := i + 1;
end;

procedure ChangePAI (var thePAI : PackedArrayIntType );
begin
end;

begin
  NextInt(5);           { Error }
  NextInt(anInt+5);     { Error }

  NextInt(PackedRec.i2); { Wrong }
  NextInt(Rec.i2);       { Right: works with an unpacked record component }

  NextInt(PackedAI[2]); { Wrong }
  NextInt(AI[2]);        { Right: works with an unpacked array component }

  ChangePAI(PackedAI);  { OK: entire packed array can be passed as a VAR parameter }
  readln(PackedAI[2]);  { OK: readln works on packed structures components }
end;
```

Constant or expression (whose ordinal value is *Value*) is out of range.

This compile time error is detected only when the Range Compile Option is on. See Chapter 13. Example:

```
var
  a : array[1..10] of 1..6;
  SmallInt : 1..10;
  Cool : green..violet;

begin
  SmallInt := 11;      { Error: 11 is too big for SmallInt }
  SmallInt := 0;       { Error: 0 is too small for SmallInt }
  SmallInt := 5 + 6;   { Error: 11 is too big for SmallInt }
  a[11] := 1;         { Error: 11 is out of range for array Index }
  a[10] := 7;         { Error: 7 is out of range for array Element }
  writeln(r : -3);    { Error: field width must be positive }
  writeln(r : 4 : 0); { Error: 2nd field width must be positive }
  Cool := red;        { Error: red is out of range for Cool.  ord(red) = 0 }

  SmallInt := 10;
  a[SmallInt + 11] := 0; { This error is not caught during compile time }
end;
```

Constant or type can't be defined in terms of itself.

See 1.7 and 3 of the Language Reference. Example:

```
const
  aConst = -aConst;      { Error }

type
  TheType = TheType;     { Error }

  RecordPtrType = ^RecordType; { OK: type pointed to is declared later }
  RecordType = record
    BadNextRecord: ^RecordType; { Wrong: can't point to defining type }
    NextRecord: RecordPtrType;  { Right: must use previously declared type }
    NestedRecord : RecordType;  { Error: recursively nested record }
  end;
```

Constant value is not numeric and must not have a sign.

See 1.7 and 5.1.2 of the Language Reference. Example:

```
const
  Dwarves = 7;
  MinusDwarves = -Dwarves;      { OK }
  MinusRed = -red;              { Error: red is non-numeric }

  MinusChar = -aCharConstant;  { Error: aCharConstant is non-numeric }
```

Control variable of FOR statement is not of ordinal type.

An ordinal type is one of integer, longint, char, or enumerated. See 3.1.1.1 and 6.2.3.3 of the Language Reference. Example:

```
for aReal := 1 to 10 do          { Error: Reals are not ordinal }
  writeln('This will not work');

for anInt := 1 to 10 do          { OK }
  writeln('anInt =', anInt);
for aLong := $10000 to $10020 do { OK }
  writeln('aLong =', aLong);
for aChar := 'a' to 'z' do       { OK }
  writeln('The letters are ', aChar);
for aColor := violet downto red do { OK }
  writeln('The colors are ', aColor);
```

Duplicate case constant in this variant-part or CASE statement.

Case constants must be unique within a case statement or record-variant. See 3.2.2 and 6.2.2.2 of the Language Reference. Example:

```
type
  BadVariantRecordType = record
    case tag : integer of
      0 : (          { OK }
        i : integer;
      );
      0 : (          { Error: second 0 case }
        r : real;
      );
    end;

begin
  case anInt of
    2 :
      writeln('two');    { OK }
    2 :
      writeln('twice');  { Error: second 2 case }
  end
end;
```

"END." is required at the end of the file.

You must have "**end.**" at the end of your file. Make sure all you **begins** are matched by **ends**. If you are puzzled, look at the Bullseye program described in Chapter 3. (See 2.1 and 8.1 of the Language Reference.)

END needed to complete the above record declaration.

See 3.2.2 of the Language Reference. Example:

```
procedure P1;
  type
    GoodRecordType = record
      aField : integer;
    end;          { Right }

    BadRecordType = record
      aField : integer;
                  { Wrong: Missing end for record declaration }
  begin
  end;
```

Execution halted.

Statement execution in the Instant Window and expression evaluation in the Observe Window will usually finish instantly (see Chapter 7). If they don't, you can halt them by clicking on the Bug Spray Can. After halting the Instant Window, the message "Execution halted" will be displayed as an Alert. After halting the Observe Window, the message "Execution halted" will be displayed in the value cell of the halted expression. Example: after a program is halted, type "*NeverReturns*" (see below) in the Observe Window and press Enter. Click on the Bug Spray Can to halt the endless loop, and "Execution halted" will be displayed in the Observe Window.

```
function NeverReturns : integer;
begin
  while true do
    ;
  end;
```

Expression can't be cast to the specified type.

When two variables are type-incompatible, they cannot be used interchangeably, even if they are the same size (as determined by *sizeof*). *Casting* bypasses this strong type-checking feature of Pascal. A cast changes a variable's type, but not its contents. For most casts, the two types must be of the same size. For example, on the MC68000 pointers and longints are type-incompatible, although both are 32 bits long. Type casting allows them to be used interchangeably.

All casting is machine dependent and potentially dangerous. Use with caution. See 5.4 of the Language Reference. Example:

```
procedure ExamplesOfCasts;
type
  LargeAType = array[1..1000] of integer;

  BigRType = record
    aComponent : LargeAType;
  end;

  SmallRType = record
    aComponent : integer;
  end;

  Packed4Type = packed array[1..4] of char;
  TwoIntType = array[1..2] of integer;

var
  Big : BigRType;
  Small : SmallRType;
  Packed4 : Packed4Type;
  TwoInt : TwoIntType;
  Large : LargeAType;
begin
  aLong := longint(aReal); { Legal, BUT just copies bits, doesn't convert to longint }
  aLong := round(aReal);   { This converts Real type to equivalent Longint type. }

  aReal := real(aLong);    { Legal, BUT just copies bits, doesn't convert to real }
  aReal := aLong;          { This converts Longint type to equivalent Real type }

  aLong := longint(Packed4); { OK: aLong and Packed4 are the same size }

  Big := BigRType(Small);   { Error: different sized records }
  Large := LargeAType(Big); { OK: Big and Large are the same size }

  Small := SmallRType(Big); { Error: different sized records }
  anInt := integer(Small);  { OK : anInt and Small are the same size }

  Large := LargeAType(Packed4); { Error: different size array }
  TwoInt := TwoIntType(Packed4); { OK: TwoInt and Packed4 are the same size array }
end;
```

Expression too complex.

Try simplifying your expression.

Field name expected. "*symbol*" isn't a field of this record.

See 4.3.2 of the Language Reference. Example:

```
var
  NotAField : integer;
  rec_1 : record
    field_1 : integer;
  end;
  REC_2 : record
    FIELD_2 : integer;
  end;
begin
  rec_1.NotAField := 0;    { Wrong: NotAField is not a field of any record }
  rec_1.field_1 := 1;     { Right }

  REC_2.field_1 := 0;     { Wrong: field_1 is a field of rec_1, not of REC_2 }
  REC_2.FIELD_2 := 1;     { Right }
end;
```

File already closed.

See 9.2.4 of the Language Reference. Example:

```
var
  f : Text;
begin
  close(f);      { Error: file was not previously opened }
  rewrite(f, 'new file');
  writeln(f, 'hello');
  close(f);      { OK }
  close(f);      { Error: file closed in previous statement }
end;
```

File already open.

A file-variable can only be used once per open file. See 9.2.3 of the Language Reference. Example:

```
var
  NEWFILE, oldFile : Text;
begin
  rewrite(NEWFILE, 'a new file');
  open(NEWFILE, 'an old file'); { Wrong: file-variable NEWFILE is already in use }
  open(oldfile, 'an old file'); { Right: use another file variable }
end;
```

File buffer not valid for write.

See 9.2.7 of the Language Reference.

```
var
  FileOfRec : file of packed record
    Field : integer;
  end;
  FileOfChar : packed file of char;
begin
  rewrite(FileOfRec, 'new file');
  put(FileOfRec);                { Error: file buffer does not have valid data }
  FileOfRec^.Field := 4;         { File buffer is now valid }
  put(FileOfRec);                { Right }
  put(FileOfRec);                { Error: file buffer no longer has valid data }
end;
```

FOR loop control variable modified. (Trap 6)

See 6.2.3.3 of the Language Reference. Example:

```
for anInt := 1 to 10 do
  anInt:= anInt+1;               { Error: modifying the control variable not allowed}
```

FOR statement control variable must be a local variable.

The Pascal Language requires for-loop control variables to be declared at the same level that they are being used. See 6.2.3.3 of the Language Reference. Example:

```
program ForLoopControlVariables;
var
  Outside : integer;             { can only be used in main program's begin-end }

procedure SubProcedure (aParam : integer);
var
  local : integer;
begin
  for integer := 1 to 10 do      { Error: "integer" isn't a variable }
  ;
  for aParam := 1 to 10 do       { Error: aParam isn't a local variable }
  ;
  for Outside := 1 to 10 do      { Error: Outside isn't a local variable }
  ;
  for local := 1 to 10 do        { OK }
  ;
end;

begin
  for Outside := 1 to 10 do      { OK: Outside is a local variable to main program }
  ;
end.
```

Formal parameter type or result type should be a named type or STRING.

When a type is used in a parameter list or in a function result declaration, it must be a named type. It cannot be an anonymous type, although this is allowed in a variable declaration. See 7.2 and 7.3 of the Language Reference. Example:

```
type
  ArrayType = array[1..2] of char;
var
  Array_1 : array[0..2] of set of char;      { OK: Array_1 is of an anonymous type }
  Array_2 : ArrayType;                      { OK: Array_2 is of a named type }
function BadArray : array[1..2] of char;    { Wrong }
begin
end;
function GoodArray : ArrayType;              { Right }
begin
end;

type
  s25Type = string[25];
function BadString25 : string[25];          { Wrong }
begin
end;
function GoodString25 : s25Type;             { Right }
begin
end;
function GoodString : string;                { Also Works }
begin
end;

type
  IntegerPtrType = ^integer;
function BadResult : ^integer;               { Wrong }
begin
end;
function GoodResult : IntegerPtrType;        { Right }
begin
end;

{ Formal parameters must also have named types }
procedure BadParam (p : ^integer);           { Wrong: ^integer is not a named type }
begin
end;
procedure GoodParam (p : IntegerPtrType);    { Right }
begin
end;
```

Formal value parameter can't be a file type or a type which contains a file type.

For every open file, there must be only one copy of the file-variable. The Operating System needs file-variables for file Input/Output. If a copy of the file-variable could be made and both were used for file I/O, then the file would be inconsistent. Therefore, passing a file-variable parameter by value (which makes a new copy) is forbidden. Passing a file-variable as a **var** parameter is permitted. See 7.3.1 of the Language Reference.

```
type
  FileOfIntType = file of integer;
  RecWithFileType = record
    FileName : string;
    FileParameter : file of real;
  end;
  ArrayOfFileType = array[1..3] of file of integer;

procedure Bad1 (F : FileOfIntType);           { Wrong: passing a filetype by value }
begin
end;
procedure Good1 (var F : FileOfIntType);      { Right: passing it as a VAR parameter }
begin
end;

procedure Bad2 (R : RecWithFileType);         { Wrong: passing a filetype by value }
begin
end;
procedure Good2 (var R : RecWithFileType);    { Right }
begin
end;

procedure Bad3 (A : ArrayOfFileType);         { Wrong: passing a filetype by value }
begin
end;
procedure Good3 (var A : ArrayOfFileType);    { Right }
begin
end;
```

FORWARD or INTERFACE procedures or functions may not later be declared INLINE.

See 7.1.3 of the Language Reference. Example:

```
unit BadInline;
interface
  function InterfaceFunc (x : char) : char; { INTERFACE declaration }
implementation
  function InterfaceFunc;
  inline                                     { Error: inline not permitted with INTERFACE }
    $4E71; { MC68000 NOP Instruction }

  procedure ForwardProc (x : char); { FORWARD declaration }
  forward;
  procedure ForwardProc;
    inline                                   { Error: inline not permitted with FORWARD }
    $4E71;
end.
```

Function name required as an actual parameter to a formal functional parameter.

Functional parameters are a way of passing functions as parameters. See 7.3.4 of the Language Reference. Example:

```
function Func (i : integer) : integer;
begin
end;
procedure Proc (i : integer);
begin
end;
procedure CallF (function aFunc (i : integer) : integer);
begin
    anInt := aFunc(4);
end;
procedure RecallF (function aFuncParam (i : integer) : integer);
begin
    CallF(Func);           { OK: CallF is passed a function }
    CallF(aFuncParam);     { OK: CallF is passed a functional parameter }
    CallF(4);              { Error: an integer is passed }
    CallF(Proc);           { Error: a procedure is passed }
end;
```

Generated code larger than 32K.

The Macintosh segment manager requires a *segment* of code to be less than 32767 bytes. In order to abide by this limitation, you must split up your file into smaller files and put them into different segments. See Chapter 5.

GOTO out of Observe/Instant not allowed.

Because of the special nature of Observe and Instant Windows, non-local *gotos*, which directly or indirectly exit a subroutine's current scope, are disallowed. For example, if you halted on the statement *writeln*; (see below) and then attempt to Observe the value of *Escape*, you will see this message in the value cell because of the *goto 9999* statement. The *goto 1* statement is a local *goto* and does not exit the *Escape*'s current scope. Furthermore, normal execution of *Escape* within your program will work without error.

```
program Escape;
label
    9999;
function Escape:integer;
label 1;
begin
    goto 1; { This GOTO is OK when Escape is called in Observe or Instant }
    anInt := 0;
1:  goto 9999; { This GOTO causes error when Escape is called in Observe or Instant }
    Escape := 42; { This statement is never executed }
end;
begin
    writeln('stop here');
    anInt := Escape; { OK to call Escape within the program }
9999:
;
end.
```

Illegal input.

When reading a value from a Text file into a variable, if the Text cannot be converted to a value for the variable type, then this runtime error occurs. See 9.4.1, 9.4.1.1, 9.4.1.2, 9.4.1.3, 9.4.1.4,, 9.4.1.5, and 9.4.2 of the Language Reference.

```
reset(TextFile, 'file containing ONLY letters');
readln(TextFile, anInt);    { Error: letters can't be converted into an integer }
readln(TextFile, aString); { OK: any ASCII text can go into a string }
```

Illegal Instruction Exception

This runtime error indicates your program has run amok. For some reason, your program has jumped off into some random place in memory, or your code has been overwritten (e.g. by an uninitialized pointer), or the Memory Manager may be corrupted. Unless you can reproduce this behavior, it may be very hard to track down the problem and to fix it. You may have to use Macsbug. (There are several bit patterns that are not legal MC68000 instructions. Your program was trying to execute one of these.)

Index out of range.

This runtime error occurs when a Pack or Unpack procedure is called with a bad index or inappropriate structures. See 10.2.1 and 10.2.2 of the Language Reference. Example:

```
var
  pac10 : packed array[1..10] of char;
  pac7  : packed array[1..7]  of char;
  ac10  : array[1..10] of char;
  ac7   : array [1..7] of char;
begin
  anInt := 1;
  pack(ac7, anInt, pac10);    { Error: ac7 is too small to fit into pac10 }
  pack(ac10, anInt, pac10);   { OK for only anInt = 1 }
  pack(ac10, anInt, pac7);    { OK for values of anInt = 1..4 }

  unpack(pac10, ac7, anInt);  { Error: pac10 is too large to fit into ac7 }
  unpack(pac10, ac10, anInt); { OK for only anInt = 1 }
  unpack(pac7, ac10, anInt);  { OK for values of anInt = 1..4 }
end;
```

InitGraf's parameter must be @thePort.

This runtime error occurs when the predefined Macintosh Toolbox call *InitGraf* is used with a parameter other than *@thePort*. This error is only detected while running with the Lightspeed Pascal environment. See *Inside Macintosh* for more details. Example:

```
InitGraf(@anInt);    { Wrong }
InitGraf(@thePort); { Right }
```

Insufficient stack space to invoke procedure or function.

This runtime error occurs when a subroutine, compiled with the Debug Compile Option, determines that there is not enough stack space remaining for it. Too many nested subroutine calls have used too much stack space for local variables. However, your heap is probably unharmed. The solution may involve fixing an out of control recursive subroutine or increasing your application's stack size in the **Run Options...** command. Use LightsBug to examine the Subroutine Call Chain. See Chapters 11 and 13. Example:

```
procedure EndlessRecursion(i:longint);
begin
  EndlessRecursion(i+1);  { Eventually, stack space will run out here }
end;
```

Integer overflow. (TRAPV)

The Overflow Compile Option inserts code that checks for intermediate arithmetic operation overflows. (See Chapter 13) This runtime error occurs when this check fails. Example:

```
anInt := 21000;
anInt := anInt + 20000;  { Error: the sum 41000 overflows anInt,   }
                        { resulting in the erroneous value -24536 }
```

INTERFACE expected here.

See 8.3 of the Language Reference. Example:

```
unit MissingKeywordINTERFACE;
{ Error: missing interface keyword here }
implementation
end.
```

Invalid case selector.

See 6.2.2.2 of the Language Reference. Example:

```
begin
  case ColorSet of
    end;

  case anInt : anInt of
    2 :
    ;
  end;
end;
```

Invalid enumerated list.

See 3.1.1.2 of the Language Reference. Example:

```
type
  BadEnum_ExtraComma = (x, y, );    { Error: extra comma (,) after last enumerated }
  BadEnumType = (0, 1, 2);          { Wrong: constants can't be used }
  GoodEnumType = (zero, one, two); { Right }
var
  BadEnumVar : (0, 1);              { Wrong }
  GoodEnumVar : (off, on);          { Right }
```

Invalid formal parameter list.

The parameter list in a function or procedure declaration is bad. See 7.3 of the Language Reference. Example:

```
procedure BadProc_1 (1, 2 : integer;    { Wrong : bad parameter names }
  ExtraSemiColon : integer;            { Wrong : extra (;) after last parameter }
);
begin
end;
procedure GoodProc_1 (one, two : integer; { Right }
  LastParameter : integer { Right }
);
begin
end;

procedure BadProc_2 (x, y :
  INTEGER var z : char ); { Wrong: missing semicolon after INTEGER }
begin
end;
procedure GoodProc_2 (x, y : INTEGER;    { Right }
  var z : char);
begin
end;

{ Wrong: aProc is declared as a procedure but it has a result type CHAR }
procedure BadCallF ( procedure aProc (i : integer) : CHAR);
begin
  aProc(5);
end;
{ Right: aProc is correctly declared as a procedural parameter }
procedure GoodCallP ( procedure aProc (i : integer));
begin
  aProc(5);
end;
{ Also Right: aFunc is correctly declared as a functional parameter }
procedure GoodCallF (function aFunc (i : integer) : char);
begin
  aChar := aFunc(5);
end;
```


Invalid INLINE constant.

Only 16-bit integer values are allowed for inline constants. See 7.1.3 of the Language Reference. Example:

```
{ Inline routine to Call a procedure through a pointer }
procedure Wrong_JSR_To(PtrToProcedure: Ptr);
inline          { Wrong: constants must be word sized }
    $205F4E90;
procedure Right_JSR_To(PtrToProcedure: Ptr);
inline          { Right: each constant is word sized }
    $205F,      { MOVEA.L    (SP)+,A0      }
    $4E90;      { JSR        (A0)          }
procedure Proc;
begin
end;
procedure Call;
begin
    Right_JSR_To(@Proc);
end;
```

Invalid label. A label must be a number between 0 and 9999.

See 1.5 of the Language Reference. Example:

```
    label
        10000;    { Error }
    label
        0, 9999;  { OK }
begin
0 :
    writeln('Legally labeled with 0');
9999 :
    writeln('Legally labeled with 9999');
end;
```

Invalid list of field names.

Field names must be legal Pascal identifiers. See 1.2 and 3.2.2 of the Language Reference. Example:

```
type
    InvalidFieldName = record
        Legal, 5, AnotherLegalName : integer;  { Error: 5 is bad field name }
    end;
```

Invalid list of variable names.

Variable names must be legal Pascal identifiers. See 4.1 of the Language Reference. Example:

```
var
    a, 1, 2, c : integer;    { Error: 1 and 2 aren't legal variable names }
```

Invalid PROGRAM parameter list.

See 8.2 of the Language Reference.

Invalid result type in function declaration.

See 7.2 of the Language Reference. Example:

```
function BadF1;                                { Wrong: missing result type }
begin
end;
function GoodF1 : char;                        { Right }
begin
end;

function BadF2 (c : char)char;                { Wrong: missing colon in result type }
begin
end;
function GoodF2 (c : char) : char;            { Right }
begin
end;

function GoodF3 : char;                        { OK: forward definition of GoodF3 }
forward;
function GoodF3;                              { OK: definition of forward function does not }
begin                                         { include its result type }
end;

{ Wrong: missing result type for formal functional parameter "aFunc" }
procedure BadCall (function aFunc (c : char));
begin
end;
{ Right }
procedure GoodCall (function aFunc (c : char) : CHAR);
begin
end;
```

Invalid variable, field, or formal parameter list. A colon (:) might be missing.

When a variable name is declared, a corresponding type must be provided. See sections 3 and 4 of the Language Reference. Example:

```
var
  VariableWithoutAType;                        { Wrong: missing type }
  Variable : integer;                          { Right }

  aRecord : record
    FieldWithoutAType;                        { Wrong: missing type }
    GoodField : integer;                     { Right }
  end;

procedure P (aParamWithoutAType;              { Wrong: missing type }
             aGoodParam : integer); { Right }
begin
end;
```

Invalid variant declaration.

See 3.2.2 of the Language Reference. Example:

```
type

BadVariantRecordType = record
  case i : integer of
    1 : (
      a, b : char;    { OK }
    );
    2 : (
      { Right: OK to have empty variant }
    );
    3 :
      ;                { Wrong: Missing () for empty variant}
  end;
```

Invalid variant selector.

See 3.2.2 of the Language Reference. Example:

```
type

BadVariantRecordType = record
  case ColorTag : ColorType+1 of    { Wrong: ColorType+1 is bad tag type }
    red: (c:char);
  end;

GoodVariantRecordType = record
  case ColorTag : ColorType of      { Right }
    red: (c:char);
  end;
```

Label has already been declared.

See 2.2.4 of the Language Reference. Example:

```
program BE_70;
  label
    6;

  procedure SubProc;
    label
      6;      { OK: This label is in a different scope }

    procedure SubSubProc;
      label
        6;    { OK: This label is in a different scope }
      label
        6;    { Error: duplicate label declaration in SubSubProc }
      begin
        6:
        ;
      end;    { SubSubProc }

    begin
      6:
      ;
    end;    { SubProc }

  begin
    6:
    ;
  end.
```

Label has already been used to label a statement.

See 2.2.4 of the Language Reference. Example:

```
procedure Proc;
  label
    13;

  begin
13 :
  writeln('OK');
13 :
  writeln('Error: there is another statement with a label 13 at this level');
end;
```

Label has not been used to label any statement.

Labels must label a statement at the level where the label is declared. See 2.1 of the Language Reference. Example:

```
program UsingLabels;

  procedure Zero;
    label
      0; { Error: This label is not used within Zero procedure, which declared it }
  begin
    writeln;
  end;

begin
end.
```

Label hasn't been declared at this level.

Statements can only be labeled with labels declared at the same level. See 2.1 and 6 of the Language Reference. Example:

```
program UsingLabels;
  label
    1;

  procedure ZeroAndOne;
    label
      0, 1;
  begin
0 :
  ;
1 :
  ;
  end; { ZeroAndOne }

begin
13 : { Error: 13 was never declared }
  writeln('Statement Labeled with 13');
0 : { Wrong: 0 was not at this level, irrelevant if 0 was declared in ZeroAndOne }
  writeln('Statement Labeled with 0');
1 : { Right: irrelevant if 1 was declared in ZeroAndOne }
  writeln('Statement Labeled with 1');
end.
```

Label was not declared at the same level as this statement.

Statements can only be labeled with labels declared at the same level. See 2.1 of the Language Reference. Example:

```
program UsingLabels;
  label
    1,2;          { declares labels for use in main program }

  procedure SubProc;
  begin
1 :               { Wrong: label 1 hasn't been declared at this level }
    goto 2;       { OK to GOTO a label at an outer scope }
  end; { SubProc }

  begin
1 :               { Right: label 1 must be used at this level }
    writeln('statement labeled with 1');
2 :               { Right: label 2 must be used at this level }
    writeln('statement labeled with 2');
  end.
```

Library must contain at least one file.

When a Project is Built into a Library, it must contain at least one file. See Chapter 9.

Library must contain only one code segment.

The Project you are trying to build into a Library contains multiple segments. Before you can build it, you must either (1) combine the segments into a single segment or (2) break up your Project into multiple single-segment Projects and build a Library for each of those. To see how your Project is segmented, click on the view control in the upper right corner of the Project Window. See Chapters 5 and 9.

Library must not contain a main program.

If a Library in a Project could have a main program, it would conflict with the Project's main program. Therefore, this is not allowed.

Line 1111 Exception

This runtime error indicates your program has run amok. For some reason, your program has jumped off into some random place in memory, or your code has been overwritten (e.g. by an uninitialized pointer), or the Memory Manager may be corrupted. Unless you can reproduce this behavior, it may be very hard to track down the problem and to fix it. You may have to use Macsbug. (On the MC68000 chip, all opcodes beginning with bits 1111 are reserved for future use. Your program is trying to execute such an opcode.)

Link Error: "symbol" is multiply defined.

- Have you included two different versions of the same Library into your Project?
- Does a Library contain an extraneous MacTraps or MacPasLib?

If you get a linker error, choose add file from the project menu, then choose cancel.

Link Error: "symbol" is undefined.

- Have you added all necessary Libraries to your project?
- Have you declared a subroutine external and never defined it in one of your Libraries?

Lower boundary of subrange is greater than upper boundary.

For types, the lower bound must be less than the upper bound. For set constructors, the lower bound can be greater than the upper bound, in which case the set is empty. See 3.1.1.3 and 5.3 of the Language Reference. Example:

```
procedure Proc;
var
  BadColor : blue..red;      { Error }
  BadSubrange : 3..1;        { Error }
  aSet : set of ColorType;

begin
  aSet := [blue..red];        { OK: aSet is empty }
end;
```

Macintosh System Error: -Number

One of the less common Macintosh errors has occurred. See 'Result Codes' in *Inside Macintosh* for more information.

Memory Full Error

The Macintosh is out of memory. This error may occur while executing a Toolbox call. See *Inside Macintosh*.

Missing or invalid unit name in this USES.

See 8.1 of the Language Reference. Example:

```
program WithBadUSES;
  uses
    4;    { Error: invalid unit name }
begin
end.
```

MODEM: or PRINTER: files must be TEXT.

The devices 'MODEM:' or 'PRINTER:' can be accessed as if they are files. If you do not open them as a Text File, you'll get this runtime error. See 9.5 of the Language Reference. Example:

```
var
  FILEOFINT : file of integer;
  FILEOFCHAR : file of char;

begin
  open(FILEOFCHAR, 'MODEM:');      { Error: can't open Modem as a File of Char }
  rewrite(FILEOFINT, 'PRINTER:');  { Error: can't open Printer as a File of Integer }

  reset(TextFile, 'MODEM:');      { OK: open Modem as a Text File }
  rewrite(TextFile, 'PRINTER:');  { OK: open Printer as a Text File }
end;
```

NIL dereference. (Trap 0)

The Range Compile Option inserts code into your program which checks for nil pointer dereferences (see Chapter 13). This runtime error occurs when this check has failed. Sometimes the actual cause of this error is long gone from the scene, but often the culprit is close at hand. You may be able to find the problem using Observe and LightsBug.

Note: if the pointer is not initialized before assignment, then you probably will not get this error. Instead, some piece of memory may be corrupted. Later, an Odd Address, an Illegal Instruction, or a Line 1111 Exception may occur.

Example:

```
{ Assumes the Range Compile Option is On }
var
  IntPtr : ^integer;
begin
  IntPtr := nil;    { It's especially important to initialize all Pointer Variables! }
  IntPtr^ := 13;    { Wrong: IntPtr is a NIL pointer }

  IntPtr := @anInt; { This makes sure that IntPtr points to something }
  new(IntPtr);      { This also, make sure that IntPtr points to something }
  IntPtr^ := 13;    { Right: IntPtr now points somewhere }
end;
```


No context

This message is displayed in the value cells of the Observe Window when your program is not running, paused, or halted. Because there is no program context, expressions (even constants) cannot be evaluated.

No windows are available for opening your file.

The maximum number of Pascal Editing Windows that can be open in Lightspeed Pascal is eight. This does not include the Instant and Observe Windows. When you receive this message, you will have to close some Windows before opening any more. Clicking on the go away box does not close a Window, it merely hides it. See Chapter 4.

Not enough memory for set operation.

During runtime, operations involving integers sets can use large amounts of memory. You can increase the heap size with the **Run Options...** command (see Chapter 6). If that does not work, you need to reimplement your program without using integer sets.

Odd Address Exception

This runtime error is the bane of all MC68000 programmers, including Macintosh programmers. You may be more familiar with it as the infamous Bomb 2. At the machine level, your program is trying to read or write a 16-bit or 32-bit quantity (which must be even-word aligned) with an odd address. On the bright side, the error is caught before it can do even more damage. Typical causes: you tried to use an uninitialized pointer or handle; the memory that was accessed through a Handle has been relocated in the Heap; you passed bad data to the Toolbox; or memory, that was important, became corrupted. Re-read *Inside Macintosh*.

Sometimes the actual cause of this error is long gone from the scene, but often the culprit is close at hand. You may be able to find the problem using Observe and LightsBug. Good luck.

Operand type incompatibility.

One man's food is another man's poison. Pascal catches statements that try to do inappropriate operations on operands. On occasion, this rule need to be broken.

Some operations aren't allowed, but it seems like they ought to be. For example, record assignments are legal because it is easy to do a blind byte copy. However, record comparisons are forbidden, because a blind byte comparison will not work, and anything else is too hard. In particular, two equivalent structures can have different bit patterns because of byte padding or different garbage in inactive variants. The solution is to write your own comparison functions for records and arrays. (Note: it is permitted to compare packed arrays of characters.)

The following examples show common errors and methods to break the rules. Use the methods at your own risk. See 3.5 and 5 of the Language Reference.

```
var
  AC1, AC2 : array[1..10] of char;
  PAC1, PAC2 : packed array[1..10] of char;
  PAI1, PAI2 : packed array[1..10] of integer;
  Rec1, Rec2 : packed record
    IntField1 : integer;
    CharField : char; { There's a byte of pad between IntField1 and IntField2 }
    IntField2 : integer;
  end;
  ptr1, ptr2 : ptr;
  string1, string2, string3 : string;
begin
  string1 := string2 + string3;           { Wrong }
  string1 := concat(string2, string3); { Right: see 10.5.3 of the Language Reference }

  aBool := (aReal <> anInt); { OK to compare real with integer }

  aBool := (aString <> AC1); { Wrong: can't compare strings with UNpacked char array }
  aBool := (aString <> PAC1); { Right: OK to compare strings with pack char array }

  anInt := 3.5 mod 4.5;      { Wrong: MOD can't take real arguments }
  anInt := 3 mod 4;          { Right }

  aColor := red + blue;      { Wrong: can't use + on enumerated types }
  aColor := ColorType(ord(red)+ord(blue)); { Right: BUT is this really meaningful? }

  aBool := (ptr1 > ptr2);     { Wrong: ptrs have no ordering }
  aBool := (longint(ptr1)>longint(ptr2)); { Right: BUT is this really meaningful? }
  aBool := (ptr1 <> ptr2);     { OK: can only use = and <> on ptrs }

  AC1 := AC2;                 { OK: it's legal to assign unpacked arrays }
  aBool := (AC1 <> AC2);       { Wrong: can't use =, <>, etc. on unpacked char arrays }
  aBool := (PAC1 <> PAC2);     { Right: OK to compare packed CHAR array }

  aBool := (PAC1 > AC1);       { Error: can't compare packed & unpacked CHAR array }
  aBool := (PAI1 <> PAI2);     { Error: can't compare packed arrays of NonChars }

  Rec1 := Rec2;               { OK to assign records }
  aBool := (Rec1 = Rec2);     { Error: can't EVER compare records this way }
end;
```

PAGE of a readonly file not allowed.

See 9.4.6 of the Language Reference.

```
reset(TextFile, 'old file');
page(TextFile);           { Error: can't write to a readonly file }
close(TextFile);
```

Parameter lists of FORWARD or INTERFACE procedures or functions can't be repeated.

Once the parameter list of a subroutine is declared in an ***interface*** part or *forward* declaration, it must not be redeclared in the subroutine's actual definition. If it were allowed, then the redeclaration of the parameter list would be redundant and could be wrong. See 7.1.1 and 8.3 of the Language Reference. Example:

```
unit aUnit;
interface
  function InterfaceFunc(c:char):char;    { INTERFACE declaration }

implementation

  function InterfaceFunc(C:CHAR):CHAR;    { Wrong: parameters declared in INTERFACE }
  begin
  end;
  function InterfaceFunc;                  { Right }
  begin
  end;

  procedure ForwardProc (c : char);        { FORWARD declaration }
  forward;

  procedure ForwardProc (C : CHAR);        { Wrong: parameter declared in FORWARD }
  begin
  end;
  procedure ForwardProc;                   { Right }
  begin
  end;

end.
```

Period (.) required following the last END of the file.

See 8.1 of the Language Reference. Example:

```
program MissingPeriod;
begin
end                                     { Error: Missing period }
```

Predefined and inline procedure names can't be used as procedural parameters.

Predefined, Toolbox, or inline calls (which aren't genuine subroutines) cannot be passed as procedural or functional parameters. However, you can write a glue routine (which wraps itself around the desired subroutine) which can be passed as a procedural or functional parameter. See 7.3.3 of the Language Reference. Example:

```
program ProceduralAndFunctionalParameters;

procedure CallF (function F (e : Extended) : integer);
begin
end;
procedure CallP ( procedure P);
begin
end;

procedure NOP;
inline
    $4e71;                { M68000 NOP instruction }
procedure NOPGlue;
begin
    NOP;
end;

function TruncGlue (e : Extended) : integer;
begin
    TruncGlue := Trunc(e);    { Predefined function }
end;

procedure OpenRgnGlue;
begin
    OpenRgn;
end;

begin
    CallF(Trunc);              { Wrong: Trunc is predefined function }
    CallF(TruncGlue);          { Right: Trunc is called from TruncGlue }
    CallP(NOP);                { Wrong: NOP is an inline procedure }
    CallP(NOPGlue);            { Right: NOP is called from NOPGlue }
    CallP(OpenRgn);            { Wrong: OpenRgn is a Toolbox procedure }
    CallP(OpenRgnGlue);        { Right: OpenRgn is called from OpenRgnGlue }
end.
```

Printer port in use by AppleTalk.

You will have to use the Modem Port instead. If you really want to use AppleTalk, see *Inside Macintosh* and Chapter 8.

Procedure name required as an actual parameter to a formal procedural parameter.

See 7.3.3 of the Language Reference. Example:

```
function Func (i : integer) : integer;
begin
end;
procedure Proc (i : integer);
begin
end;
procedure CallP ( procedure aProc (i : integer));
begin
    aProc(4);
end;

procedure RecallP ( procedure aProcParam (i : integer));
begin
    CallP(Proc);           { OK: CallP is passed a procedure }
    CallP(aProcParam);     { OK: CallP is passed a procedural parameter name }
    CallP(4);              { Error: an integer is passed }
    CallP(Func);           { Error: a function is passed }
end;
```

Procedure or function has an INTERFACE or FORWARD declaration but was never defined.

The subroutine declarations that are in the *interface* part or use the *forward* directive allow you to defer the subroutine's definition. However, the definition must appear later in the file. See 7.1.1 and 8.3 of the Language Reference. Example:

```
unit aUnit;
interface
    procedure aProc;           { Error: never defined later in the IMPLEMENTATION part }
implementation
    procedure aForward;        { Error: never defined later in this unit. }
    forward;
end.                           { end of unit }
```

PROGRAM or UNIT is missing from the beginning of this file.

You must have the keyword **program** or **unit** at the beginning of your file. If you are puzzled, look at the Bullseye program described in Chapter 3. (Also see 8.1 of the Language Reference.)

Project has a code segment larger than 32K.

The code segment must be broken up into smaller segments, each having a size of less than 32767 bytes. View the project by segment and then drag one or more files to a different (or new) segment. See Chapter 5.

Project has more than 32K bytes of globals.

Unfortunately, the Macintosh/MC68000 architecture doesn't allow an application to directly have more than 32K bytes of globals. However, you can indirectly use more memory with storage allocation subroutines. See 10.1 and 10.2 of the Language Reference. For large data structures, the use of Handles is highly recommended. See *Inside Macintosh*.

Project has no main program.

In order for your program to run, a file which contains a main program needs to be added to your Project. The file should contain the **program** keyword. If you are puzzled, look at the Bullseye program described in Chapter 3.

Read from a writeonly file not allowed.

Files opened with *rewrite* are WriteOnly. See 9.3.1 and 9.4.1 of the Language Reference. Example:

```
REWRITE(TextFile, 'WriteOnly File');
get(TextFile);                      { Error }
read(TextFile, anInt);              { Error }
readln(TextFile, anInt);            { Error }
```

Read or write of an unopened file not allowed.

This run time error occurs when any input or output operation is done using a file-variable that has not been previously opened. See Section 9 of the Language Reference. Example:

```
var
  TextFile : Text;
begin
{ All these calls give an error because a file was never opened with TextFile }
  reset(TextFile);                  { Error: Assumes a file was already open }
  write(TextFile, anInt);           { Error }
  read(TextFile, anInt);            { Error }
  page(TextFile);                   { Error: page does an implicit write to a file }
  get(TextFile);                    { Error: get does an implicit read of a file }
end;
```

READLN and WRITELN can only be used with text files.

The predefined procedures *readln* and *writeln* only work with Text files. Use *read*, *write*, *put* or *get* for all other types of files. See 9.3 and 9.4 of the Language Reference. Example:

```
procedure Proc;
var
  FileOfChar : file of char;           { not the same thing as Text }
  PackedFileOfChar : packed file of char; { not the same thing as Text }
  FileOfInt : file of integer;         { not at all the same as Text }

begin
  open(TextFile, 'Text File');
  writeln(TextFile, 4);                 { OK }
  readln(TextFile, anInt);              { OK }

  open(PackedFileOfChar, 'packed file of CHAR');
  writeln(PackedFileOfChar, 'a')        { Wrong }
  write(PackedFileOfChar, 'a');         { Right }

  readln(PackedFileOfChar, aChar);      { Wrong }
  read(PackedFileOfChar, aChar);        { Right }

  open(FileOfChar, 'file of CHAR');
  writeln(FileOfChar, 'a')              { Wrong }
  write(FileOfChar, 'a');               { Right }

  open(FileOfInt, 'file of INTEGERS');
  writeln(FileOfInt, 4);                { Wrong }
  write(FileOfInt, 4);                 { Right }

end;
```

Record contains no fields.

Records must contain at least one field. Example:

```
type
  RecordType = record

end;                                { Error: no fields }
```

Record name expected. "symbol" isn't a record.

Symbol is not properly used in a record reference. See 6.2.4 of the Language Reference.

```
type
  RecordType = record
    aField : integer;
  end;
  RecordPtrType = ^RecordType;
var
  aRecord : RecordType;
  aRecordPtr : RecordPtrType;

function aFunc : RecordPtrType;
begin
end;

begin
  anInt.aField := 3;      { Wrong: anInt is not a record }
  aRecord.aField := 3;    { Right }
  with anInt do           { Wrong: anInt is not a record }
    aField := 3;
  with aRecord do         { Right }
    aField := 3;
  new(aRecordPtr);        { aRecordPtr must point to something before it can be used }
  aRecordPtr.aField:=3;    { Wrong: need to dereferenced a ptr before it can be used }
  aRecordPtr^.aField:=3;  { Right: the ^ makes all the difference }

  aFunc^.aField := 3;     { Wrong: Legal in LisaPascal, but not in Lightspeed Pascal }
  aRecordPtr := aFunc;    { Right:... }
  aRecordPtr^.aField := 3; { ... a Temporary is required }
end;
```

Result can't be assigned to the function named "symbol" outside of its declaration.

See 7.2 of the Language Reference. Example:

```
program AssigningToFunctions;
type
  IntPtrType = ^integer;
var
  IntPtr : IntPtrType;
function aFunction : integer;
begin
  aFunction := 42;      { Right: this is how results are returned from functions }
end;
function ReturnIntPtr : IntPtrType;
begin
end;
begin
  aFunction := 42;      { Error: outside of function declaration }

  ReturnPtrToInt^ := 4; { Wrong: Legal in LisaPascal, but not in Lightspeed Pascal }
  IntPtr := ReturnIntPtr; { Right: ... }
  IntPtr^ := 4;        { ... A Temporary variable is needed }
end.
```


SANE Floating Point Error

A SANE (Standard Apple Numerics Environment) call has resulted in an error. Note: a Floating point division by zero results in this error, while an integer-type division by zero results in the Zero Divide Exception error. See Appendix D. Example:

```
anExt := sqrt(-4.0); { Error: square root of a negative number }
anExt := anExt / 0.0; { Error: Floating Point Division by Zero. }
```

SEEK of a device not allowed.

Devices can often be treated as if they are files. However, *seek* cannot be used with devices. Example:

```
open(FileOfText, 'MODEM:');
seek(FileOfText, 0);          { Error }
```

SEEK of a negative component number not allowed.

See 9.2.8 of the Language Reference. Example:

```
open(FileOfInt, 'old file');
seek(FileOfInt, -1);          { Error }
seek(FileOfInt, 0);           { OK: seeks to first component (an integer) in the file }
seek(FileOfInt,maxlongint);   { OK: seeks to last component (an integer) in the file }
```

SEEK of a readonly or writeonly file not allowed.

Only files opened with the procedure *open* are ReadWrite. The procedure *seek* can only be used with ReadWrite files. See 9.2.8 of the Language Reference. Example:

```
rewrite(FileOfInt, 'new file');
seek(FileOfInt, 0);           { Error }
close(FileOfInt );
reset(FileOfInt, 'old file');
seek(FileOfInt, 0);           { Error }
close(FileOfInt );
open(FileOfInt, 'old file');
seek(FileOfInt, 0);           { OK }
close(FileOfInt );
```

Segment Loader error (not enough memory).

When a subroutine in an unloaded segment is called, the segment is automatically loaded into the heap. This error will occur when there is not enough memory to load the segment. You can either increase your Heap size with the **Run Options Command...**, or use *UnloadSeg* to unload unneeded segments. See *Inside Macintosh*.

Selector expression of CASE statement isn't of ordinal type.

See 3.1.1 and 6.2.2.2 of the Language Reference. Example:

```
begin
  case aReal of      { Error: aReal is not ordinal }
    3.14159 :
      ;
  end
  case ColorSet of   { Error: ColorSet is not ordinal }
    [red] :
      ;
  end
  case anInt of      { OK: anInt is ordinal }
    0 :
      ;
  end;
  case aChar of      { OK: aChar is ordinal }
    'c' :
      ;
  end;
  case aColor of     { OK: aColor is ordinal }
    red :
      ;
  end;
  case aLong of      { OK: aLong is ordinal }
    $40000 :
      ;
  end;
end;
```

Semicolon (;) or END expected after the previous statement.

See 6.2.1 of the Language Reference. Example:

```
begin
  writeln('Error: Missing SemiColon at end of this line -->')
  writeln('OK: SemiColon not needed at end of this line')
end;
```

Semicolon (;) or UNTIL expected after the previous statement.

See 6.2.3.1 of the Language Reference. Example:

```
begin
  repeat
    writeln('OK');
  until false;

  repeat
    writeln('Error: missing semicolon on this line -->')
    writeln
  until false;

  repeat
    writeln('Error: Missing UNTIL');
  until false;

end;
```

Semicolon (;) required on this line or above.

See 2.1, 1.7, 3, 4.1, 7.1,7.2 of the Language Reference.

```
label
  0                                { Error: missing semicolon at end of line }
const
  BadConstant = 0                  { Error: missing semicolon at end of line }
type
  BadType = char                   { Error: missing semicolon at end of line }
var
  BadVar : integer                 { Error: missing semicolon at end of line }
procedure BadProc : CHAR;         { Error: procs do not return values }
begin
end;
function BadFunc : char           { Error: missing semicolon at end of line }
begin
end;
```

Set elements must be integer, char or enumerated.

Set Element types are limited. However, in Lightspeed Pascal the ranges are not as limited as in other Pascal Implementations. See 3.2.2 of the Language Reference. Example:

```
type
  ArrayType = array[1..3] of integer;
  RecordType = record
    aField : integer;
  end;
  SetType = set of char;
var
  SetOfArray : set of ArrayType;    { Error }
  SetOfLongint : set of longint;    { Error }
  SetOfRecord : set of RecordType;  { Error }
  SetOfReal : set of real;          { Error }
  SetOfSet : set of SetType;        { Error }

  SetOfColor : set of ColorType;    { OK }
  SetOfChar : set of char;          { OK }
  SetOfSmallInts : set of 1..10;    { OK }
  SetOfintegers : set of integer;   { OK }
```

Set value out of range. (Trap 3)

When a set assignment occurs, the range of the resulting set expression must be within the range of variable's set type. Example:

```
var
  DigitSet : set of 0..9;
  WarmSet : set of red..yellow;
begin
  DigitSet := [-1, 1];           { Error: -1 not in range 0..9 }
  DigitSet := [5..10];          { Error: 10 not in range 0..9 }

  DigitSet := [];               { OK }
  DigitSet := [0..9];           { OK }
  DigitSet := [1, 2, 3, 5, 7];  { OK }
  if 1000 in DigitSet then      { OK: even if 1000 isn't in range 0..9 }
  ;
  DigitSet := DigitSet - [10..15]; { OK: expression "DigitSet-[10..15]" is in range }
  WarmSet := [blue];            { Error: blue not in range red..yellow }

  WarmSet := [];               { OK }
  WarmSet := [red..yellow];     { OK }
end;
```

Stack Ran Into Heap Error

The stack has overflowed into the heap; the heap is probably damaged. This error, which may be more familiar as Bomb 28, is not always detected. (See *Inside Macintosh*.) In hindsight, you should have compiled with the Debug Option, which probably would have caught this error safely and in time. (See Chapter 13.)

String constant must have length 1 to be compatible with Char.

See 3.5.3 of the Language Reference Manual.

```
const
  aStringConstant = 'Hi';
  NullStringConstant = '';
  CouldBeEither = '?';
begin
  aChar := '';                 { Error }
  aChar := NullStringConstant; { Error }
  aChar := 'Hi';               { Error }
  aChar := String2Constant;    { Error }

  aChar := 'C';                { OK }
  aChar := CouldBeEither;      { OK }
end;
```

String constant too large or too small for destination

When assigning a string constant to a string variable, the string constant must fit. When assigning a string constant to a packed array of characters, the length of the string constant must be the same as the number of characters in the array. Example:

```
var
  s5: string[5];
  PAC5 : packed array[1..5] of char;
begin
  s5 := 'twelve chars';    { Error: the length of the string constant > 5 }
  s5 := 'five!';           { OK }
  PAC5 := 'twelve chars'; { Error: the length of the string constant <> 5 }
  PAC5 := 'four';          { Error: the length of the string constant <> 5 }
  PAC5 := 'five!';        { OK }
end;
```

String range error. (Trap 2)

Do not confuse a string's size with its length. This runtime error is caused by accessing a character element of a string with an index outside the range `1..length(aString)`. If you need to lengthen a string, use the *insert* procedure or *include* function. See 3.3, 4.3.1, and 10.5 of the Language Reference. Example:

```
aString := 'foo';
aString[3] := 'u';    { OK: current length of aString is 3 }
aString[4] := 'r';    { Wrong: can't append to end of a string using indexing }
insert('r', aString, 4); { Right: aString = 'four' }
```

String size must be a number between 1 and 255.

See 3.3 of the Language Reference. Example:

```
var
  aStringFloating : string[3.4]; { Wrong: string size not an integer }
  aString256 : string[256];      { Wrong: string size > 255 }
  aStringMinusOne : string[-1];  { Wrong: string size < 1 }

  aString5 : string[5];          { OK }
```

String too large.

If a string is being read into by a *read*, *readln*, or *readstring* statement, the destination string must be big enough to accommodate the result. See 9.4.1.4 of the Language Reference.

```
var
  aString5 : string[5];
begin
  readstring('The input string has more than 5 chars', aString5); { Error }
end;
```

Subrange boundaries are not of the same type.

Two constants in a subrange definition must have compatible types. See 3.1.1.3 of the Language Reference. Example:

```
var
  SubrangeOfDifferentTypes : red..HEAVY;    { Wrong }
  GoodSubRange : red..yellow;                { Right }
  BadSet : set of red..HEAVY;                { Wrong }
  GoodSet : set of red..blue;                { Right }
  AlsoGoodSet : set of ColorType;           { Also Right }
```

Subrange boundary is not integer, char, or enumerated.

See 3.1.1.3 of the Language Reference.

```
type
  BadLongintType = -100000..200000; { Error: longint in subrange is disallowed }
  BadRealType = 2.5..3.1;           { Error: real in subrange is disallowed }
var
  LongintSubrange : -100000..200000; { Error: longint in subrange is disallowed }
  RealSubrange : 2.5..3.1;           { Error: real in subrange is disallowed }
```

Subrange error. (CHK)

Subrange error. (Trap 1)

These errors mean the same thing. The Range Compile Option inserts code which checks for out of range errors in your program (see Chapter 13). This runtime error occurs when this check has failed. Example:

```
var
  Small : 1..5;
  Big : -20000..20000;
begin
  anInt := 6;
  Small := anInt;      { Error }
  Small := anInt - 3;  { OK }

  anInt := 30000;
  Big := anInt;        { Error }
end;
```

Tag type must be ordinal.

See 3.1.1 and 3.2.2 of the Language Reference.

```
type
  BadVariantRecordType = record
    case FloatingPointTag : Real of
      3.1416 : (           { Wrong: tag type can't be Real }
        Pi : string[2]
      );
    end;
  GoodVariantRecordType = record
    case i : integer of
      3 : (           { Right }
        Pi : string[2]
      );
    end;

  OKVariantRecordType = record
    case longTag : longint of
      $12345 : (       { OK: tag type can be Longint }
        L : string[8]
      );
    end;
```

The entire read will be re-executed when you continue.

If you click on the Bug Spray Can while reading from the Text Window with *read* or *readln*, the read will stop and all pending input will be thrown away. When you continue the program, the entire read will be started over again.

This declaration or statement doesn't belong here.

See Section 8 of the Language Reference. Example:

```
unit BadUnit;
interface
  label           { Error: Can't have a label here }
  0;
implementation
  label           { Error: Can't have a label here }
  1;
end.
```

This doesn't make sense.

This error covers a multitude of sins, not all of which are described here. This error occurs when some piece of the Pascal code does not fit the "railroad" syntax diagrams. Often, the text, following the error on the same line, is outlined. See the Language Reference. Example:

```
program;    { Error: missing program name. See 8.1 of the Language Reference }
type
  StringType = string;
var
  BadArray : array[5] of char;      { Wrong: doesn't declare an array of 5 chars }
  GoodArray : array[1..5] of char;  { Right }

  PointerToString : ^string;        { Wrong: Can't use STRING in an anonymous type }
  PointerToString : ^StringType;    { Right: See 3.4 of the Language Reference }

   $\pi$  : Extended;      { Wrong: can't use special Characters for Pascal identifiers }
  Pi : Extended;      { Right: See 1.2 of the Language Reference }
procedure ProcWithoutArgs;
begin
  aChar := ' $\pi$ ';      { OK: can use Macintosh Character set for strings & comments }
end;

begin
  anInt :=;           { Error: missing value }
  ProcWithoutArgs(); { Wrong: Procs and Funcs without args mustn't have (), unlike C }
  ProcWithoutArgs;   { Right: It does look funny if you are used to the C Language }
  case anInt of
    11 writeln('Error: missing colon after 11, see 6.2.2.2' );
  end;

  if true then
    writeln('Error: semicolon does not belong at the end of this line ->');
  else { See 6.2.2.1 of the Language Reference }
    writeln('OK: the problem is with the previous writeln');

  writeln('Error: Missing close quote -> );
  Error missing open comment }
  { Error missing close comment
  if (p <> nil) & (p ^ = 1) then { See Appendix B }
    writeln('Error: Short-circuit Boolean not allowed in Lightspeed Pascal' );
end.
```


This doesn't make sense as a statement.

Everything between a **begin** and an **end** must be a statement. See 2.1 and 6 of the Language Reference. Example:

```
program NonStatements;
begin
  { The following are illegal because they aren't statements. }
  inline
  string;
  aBool = false;    { This is an expression, not an assignment }
  const
  unit bad;
  var
  type
  const
end.
```

This file along with its used interfaces is too large to be compiled.

A file may not exceed approximately 2500 lines, including the **interface** parts of any units the file **uses**. See Chapter 8 for a discussion on units.

This file USES too many units.

You can have a large number of units in your Project. However, there is a maximum number of units a file can use. Try to eliminate unnecessary units in the uses-clause. See Chapter 8 for a discussion on units.

This is not allowed in the Instant window.

The Lightspeed Pascal Instant Window is very special. It can execute any Pascal statement that would be legal at the point where your program is currently halted, except for **gotos**. See Section 6 of the Language Reference.

Any new declarations are disallowed. You might want to declare some scratch variables in your main program for use in the Instant Window.

This statement or keyword doesn't belong here.

Possible misplaced or missing keywords. See 8.1, 2.1, and 8.3 of the Language Reference. Example:

```
program BadProgram;
writeln('BAD');    { Error: statements don't go in declaration part }
i : integer;       { Error: Need VAR before declaring variables }
type
  t = integer;
end;               { Error: END doesn't go after TYPE declaration }
begin
end. { End of BadProgram }
```

Another example:

```
unit BadUnit;
interface
  label
    13;           { Error: labels can't be made visible outside the unit }
implementation
  label
    13;           { Error: labels can't be shared inside the unit }
  procedure Proc;
  begin
13:               { Error: 13 needs to be declared as a label inside this proc }
  ;
  end;
end. { End of BadUnit }
```

Too few parameters used in procedure or function call.

See 7.3 of the Language Reference.

Too many constants in enumerated type.

For space efficiency, enumerated types are implemented as unsigned byte values ranging from 0 to 255. Therefore, an enumerated type can have a maximum of 256 enumerated constants.

Too many indices are being applied to a variable or expression.

Array elements have to be accessed the same way in which they are declared. See 4.3 and 4.3.1 of the Language Reference. Example:

```
var
  OneDim : array[1..10] of integer;
  TwoDim : array[1..10, 1..10] of integer;
begin
  OneDim[1][2] := 3;      { Error }
  OneDim[1, 2, 3, 4] := 4; { Error }
  OneDim[3] := 5;         { OK }
  TwoDim[1][2] := 3;      { OK }
  TwoDim[1, 2] := 4;      { OK: Pascal allows either form }
end;
```

Too many parameters used in procedure or function call.

See 7.3 of the Language Reference.

Too many up-arrows (^) are being applied to a variable or expression.

Too many up arrows result in too many dereferences. (If too few up arrows are applied, a type incompatibility error often results). See 4.3 and 4.3.4 of the Language Reference.

```
type
  pType = ^integer;
  ppType = ^pType;
var
  p : pType;
  pp : ppType;
  i : integer;
begin
  anInt := p^^;      { Wrong }
  anInt := p^;       { Right }

  anInt := pp^^^;    { Wrong }
  anInt := pp^^;     { Right }
end;
```

Type expected. "symbol" isn't a type.

See sections 3 and 4 of the Language Reference. Example:

```
var
  NotAType : integer;
  s : set of NotAType;      { Error: NotAType is not a type }
  i : NotAType;             { Error: NotAType is not a type }
  BadArray : array[aConstant] of char;      { Wrong: needs a type or subrange }
  GoodArray : array[1..aConstant] of char; { Right }
type
  aBadType = NotAType;      { Wrong }
  aGoodType = integer;      { Right }
begin
end;
```

Type incompatibility.

You can't add apples and oranges. Pascal catches illogical statements which try to use incompatible types. On occasion, this rule need to be broken. The following examples show common errors and methods to break the rules. Use the methods at your own risk. Type compatibility can be subtle. See 3.5.2 and 5 of the Language Reference. Example:

```
var
  PtrToInt : ^integer;
  PtrToChar : ^char;
begin
  aBool := (aColor = 1);           { Wrong: can't compare incompatible types }
  aBool := (Ord(aColor) = 1);       { Right: but what do you really want to do? }
  aBool := (aColor = ColorType(1)); { Also Right: but what do you really want to do? }
  aBool := (aString = aChar);       { OK to compare strings to char }
  aBool := (aString = anInt);        { Wrong: strings and integers aren't compatible }
  aBool := (aString = stringof(anInt : 1)); { Right: anInt is converted to a string }
  aBool := (red in aColor);          { Wrong: aColor isn't a set }
  aBool := (red in [red..violet]);  { Right }
  aBool := (2 in [red..violet]);    { Wrong: 2 isn't in set of ColorType }
  aBool := (red in [red..violet]);  { Right }
  aBool := ('a' in [$OD..'z']);      { Wrong: $OD is a different type from 'z' }
  aBool := ('a' in [chr($OD)..'z']); { Right: chr($OD) is carriage return }
  aBool := (PtrToInt = PtrToChar);  { Wrong: pointers are of different types }
  aBool := (pointer(PtrToInt) = PtrToChar); { Right: but what do you really want to do? }
end;
```

Type incompatibility between an actual and formal procedural or functional parameter.

The functional or procedural parameter list must match the parameter list declared in the called subroutine. See 7.3.5 of the Language Reference. Example:

```
procedure P1 (r : real);
begin
end;
procedure P2 (x, y : integer);
begin
end;
procedure CallP (procedure aProc (i : integer));
begin
  aProc(4);
end;

function F (i : integer) : real;
begin
end;
procedure CallF (function aFunc (i : integer) : integer);
begin
  writeln(aFunc(4));
end;

begin
  CallP(P1); { Error: P1 doesn't have same parameter types as aProc }
  CallP(P2); { Error: P2 doesn't have same number of arguments as aProc }
  CallF(F);  { Error: F doesn't have same result type as aFunc }
end;
```

Type incompatibility between an actual and formal value parameter.

Pascal requires assignment compatibility between *actual* value parameters (the actual expressions passed to a subroutine) and *formal* value parameters (the arguments in the subroutine definition). Assignment compatibility of types can be subtle. See 7.3.1 and 3.5.3 of the Language Reference. Example:

```
type
  CanonicalType = array[1..4] of char; { a type }
  IdenticalType = CanonicalType;      { an identical type with CanonicalType }
  AnotherType = array[1..4] of char;  { a type DIFFERENT from CanonicalType }
var
  AnonArray : array[1..4] of char;    { an array of an anonymous type }
  CanonicalArray : CanonicalType;      { a variable of a named type }
  IdenticalArray : IdenticalType;     { a variable of a same named type }
  AnotherArray : AnotherType;         { a variable of a different named type }
  PAC : packed array[1..6] of char;

procedure RealValue (r : real);
begin
end;
procedure IntValue (i : integer);
begin
end;
procedure ArrayValue (a : CanonicalType ); { type of 'a' must be named }
begin
end;

begin
  RealValue(aReal);           { OK }
  RealValue(anExt);           { OK: anExt gets converted to real }
  RealValue(anInt);           { OK: anInt gets converted to real }
  RealValue(aLong);           { OK: aLong gets converted to real }

  IntValue(aLong);            { OK: aLong gets converted to integer }
  IntValue(aColor);           { Error: enums don't get converted to integer }
  IntValue(aReal);            { Wrong: reals don't get converted to integer }
  IntValue(round(aReal));      { Right }
  IntValue(aPtr);             { Error }

  anInt := length('string');  { OK }
  anInt := length(aChar);     { OK: aChar is treated as a string }
  anInt := length(PAC);       { OK: PAC [1..N] is treated as a string }
  anInt := length(AnonArray); { Error: unpacked arrays don't get converted }

  { These examples are subtle - see 3.5.1 of the Language Reference }
  ArrayValue(CanonicalArray); { OK }
  ArrayValue(IdenticalArray); { OK: the types are identical }
  ArrayValue(AnonArray);      { Error: anonymous types aren't identical with anything }
  ArrayValue(AnotherArray);   { Error: the named types are different }
end;
```

Type incompatibility between an actual and formal VAR parameter.

Pascal requires the types of actual **var** parameters (the actual variable passed to a subroutine) and the types of formal **var** parameters (the arguments in the subroutine definition) to be identical. When a parameter is passed to a formal **var** parameter, the called subroutine can change the actual parameter passed, not just a copy of it. Therefore, the data representation of the actual parameter must exactly match the data representation of the formal parameter. Type identity can be subtle. See 7.3.2 and 3.5.1 of the Language Reference.

```
type
  CanonicalType = array[1..4] of char; { a type }
  IdenticalType = CanonicalType;      { an identical type with CanonicalType }
  AnotherType = array[1..4] of char;  { a type DIFFERENT from CanonicalType }
var
  AnonArray : array[1..4] of char;    { an array of an anonymous type }
  CanonicalArray : CanonicalType;      { a variable of a named type }
  IdenticalArray : IdenticalType;      { a variable of a same named type }
  AnotherArray : AnotherType;          { a variable of a different named type }
  PAC : packed array[1..6] of char;

procedure RealVar (var r : real);
begin
end;
procedure IntVar (var i : integer);
begin
end;
procedure StringVar (var s : string);
begin
end;
procedure ArrayVar (var a : CanonicalType ); { type of 'a' must be named }
begin
end;

begin
  RealVar(aReal);           { OK }
  RealVar(anExt);           { Error: types must match exactly for VAR }
  RealVar(anInt);           { Error }
  RealVar(aLong);           { Error }

  IntVar(anInt);            { OK }
  IntVar(aColor);           { Error }
  IntVar(aLong);            { Error }
  IntVar(aReal);            { Error }
  IntVar(aString);          { Error }
  IntVar(aPtr);             { Error }

  StringVar(aString);       { OK }
  StringVar(aString25);     { OK: any string type is identical with type VAR s:STRING }
  StringVar(aChar);         { Error }
  StringVar(PAC);           { Error }
  StringVar(AnonArray);     { Error: unpacked char arrays don't match strings }

  { These examples are subtle - see 3.5.1 of the Language Reference }
  ArrayVar(CanonicalArray); { OK }
  ArrayVar(IdenticalArray); { OK: the types are identical }
  ArrayVar(AnonArray);      { Error: anonymous types aren't identical with anything }
  ArrayVar(AnotherArray);   { Error: the named types are different }
end;
```

Type needed to complete a declaration on this line or above.

Whenever a variable or record field is declared, its type must accompany the declaration. See 4.1 and 3.2.2 of the Language Reference. Example:

```
var
  aRecord : record
    BadField : ;      { Wrong: missing type }
    GoodField : CHAR; { Right }
  end;
  BadVariable : ;      { Wrong: missing type }
  GoodVariable : CHAR; { Right }
begin
end;
```

Type or procedure name used where a variable, field name, or value is required.

See Sections 5 and 6 of the Language Reference. Example:

```
type
  aType = integer;
  aName = integer;
var
  aRecord : record
    aName : integer
  end;

procedure aProc;
begin
end;
begin
  anInt := aName;      { Error: aName refers to a type in this statement }
  if anInt = aType then { Error: aType is not a value }
  ;
  with aRecord do
    begin
      anInt := aName;    { OK: aName refers to field in aRecord in this statement }
      anInt := aProc;    { Error: aProcedure is not a value }
      anInt := aType;    { Error: aType is not a value }
    end;
  end;
end;
```

Unexpected end of file.

EOF (End Of File) was reached while doing a Read. You should always check for EOF before reading. See 9.9 of the Language Reference. Example:

```
RESET(FileOfText, 'input file');
while true do      { Error: eventually tries to read past end of file }
  readln(FileOfText, aString);
close(FileOfText);
RESET(FileOfText, 'input file');
while not eof(FileOfText) do { Right: checks for EOF before reading from file }
  readln(FileOfText, aString);
```

USES only allowed immediately after PROGRAM heading or INTERFACE.

See 8.1 and 8.3 of the Language Reference. Example:

```
program WhichUsesUnits;
  uses
    aUnit;          { Right }
  var
    i : integer;
  uses
    AnotherUnit;    { Wrong: USES must go after program or interface heading }

begin
end.
```

Value out of range.

The range of the value read is checked before it is placed into a variable. This runtime error occurs when the value read falls outside of range of the variable's type. Example:

```
var
  Small: 1..5;
begin
  readstring('100', Small);    { Error }
  readstring('100000', anInt); { Error }
end;
```

Variable or function name expected. "symbol" isn't a variable or function.

The target of an assignment must be a variable or a function name when a return result is being set. See 6.1.1 and 7.2 of the Language Reference.

```
type
  NotAVariable = integer;
var
  aVariable : integer;
  aPtrToInteger : ^integer;
procedure aProcedure;
begin
end;

function aFunction : integer;
begin
  NotAVariable := 4;    { Wrong: NotAVariable is not a variable }
  aVariable := 4;       { Right }
  aPtrToInteger^ := 4;  { Also Right }

  aProcedure := 4;      { Wrong: aProcedure is not a function name }
  aFunction := 4;       { Right: this is how functions return values in Pascal }
end;
```


Variables of a file type or a type which contains a file type can't be assigned.

For every open file, there must only be one copy of the file-variable. The Operating System needs file-variables for file Input/Output. If a copy of the file-variable could be made, and both were used for file I/O, then the file would be inconsistent. Therefore, making a new copy of a file-variable is forbidden. Example:

```
var
  file_1, file_2 : file of integer;
  array_1, array_2 : array[1..3] of file of integer;
  record_1, record_2 : record
    FileName : string;
    FileParameter : file of real;
  end;
begin
  open(file_1, 'My file');
  file_1 := file_2;    { Error: could cause havoc with filesystem ... }
  close(file_2);       { ...especially, if this was done }

  array_1 := array_2; { Error: these arrays contain filetypes }
  record_1 := record_2; { Error: these records contain filetypes }
end;
```

Variables of this type would be too large.

Because Lightspeed Pascal use 16 bit offsets (the MC68000 architecture doesn't allow 32 bit offsets), a data structure cannot exceed 32766 bytes.

```
type
  t = packed array[1..32767] of char; { Error: type is too big to be defined }
var
  a : packed array[1..32767] of char; { Error: an array of this size is too big }
```

Variant-part of record contains no variants.

See 3.2.2 of the Language Reference. Example:

```
type
  BadVariantRecordType = record
    case tag : integer of
      --
    end;
    { Error: must have at least one variant here }
```

WARNING: The file "filename " has been modified outside of this project.

To automate the build process (and reduce the amount of tedious work you would have to do), the Project keeps careful track of the source and Library files which it uses. You will get this notification when you build a Project, or open or load a file, in which the file's last date-time modification is different from the one the Project remembers.

While your program is halted, you may only DRAG your program's windows.

While your program is halted, your code, which handle events in your program's windows, is inactive. Therefore, the Lightspeed Pascal Environment only allows you to drag your windows.

WITH statement too deeply nested.

In this implementation of Pascal, **with** statements are restricted to a depth of 9 record references. Example:

```
var
  r1, r2, r3, r4, r5, r6, r7, r8, r9, r10_TOO_DEEP : record
    aField : integer;
  end;
begin
  with r1, r2, r3, r4, r5, r6, r7, r8, r9 do
    with r10_TOO_DEEP do { Error: more than 9 record references }
      begin
        end;
      end;
    end;
  end;
```

Write to a readonly file not allowed.

Files opened with *reset* are *ReadOnly*, and therefore, you cannot write to them. See 9.2, 9.3.1, and 9.4.1 of the Language Reference.

```
RESET(FileOfText, 'old file');
put(FileOfText); { Error }
writeln(FileOfText, 'Error: cannot writeln to ReadOnly file');
```

Writing over a Project or Library is not allowed.

This is a safety feature which prevents you from overwriting a Project or Library by accident. You can delete the Project or Library by using the **Delete** Command in the **File** Menu, and then re-use the filename.

You already have a window named "*Name1*". Would you like to open your file as "*Untitled N*"?

You may have tried to open a file twice. You may have two different files with the same, name but in different volumes or HFS folders. At this point, you can either open the file and see what's in it, or you can cancel.

You can't RESET a nonexistent file.

When *reset* is called with a file name, that file must already exist in order to read from it. See 9.2.1 of the Language Reference. Example:

```
reset(FileOfText, 'Non Existent File'); { Error }
```

You can't RESET(OUTPUT) or REWRITE(INPUT).

A *reset* of the predefined file *output* means that you are trying to redirect *output* for reading. A *rewrite* of the predefined file *input* mean that you are trying to redirect *input* for writing. Neither of these make any sense. See 9.1 of the Language Reference.

```
program InputAndOutput(input, output);
begin
  reset(output);      { Wrong }
  reset(input);       { Right }
  rewrite(input);     { Wrong }
  rewrite(output);    { Right }
end.
```

Zero Divide Exception

An integer-type division resulted in a division by zero. If the division was floating point, the "SANE Floating Point Error" would occur instead. Example:

```
anInt := anInt div 0;
```

Appendix G

Macsbug Reference

Introduction	G-3
Using Macsbug	G-3
Command Syntax	G-4
Program Execution Control	G-4
Set and Display Commands	G-4
Expressions	G-5
A-Trap Commands	G-5
Program Termination	G-6
Heap Zone Commands	G-6
Miscellaneous Commands	G-7
Handy Hints	G-8

Macsbug Reference

Introduction

The Macsbug family of Macintosh debuggers gives the programmer visibility into and control of executing Macintosh programs. The debugger resides on the same machine as the executing program. Macsbug allows the programmer to interrupt and resume the program, set and clear breakpoints, and display memory in a variety of formats. The debugging information is displayed on the executing program's Macintosh screen (temporarily replacing the program's screen image).

Macsbug uses about 40K of memory when it's installed. It uses about 18K of disk space.

Macsbug takes control under the following circumstances:

- 1) When the programmer interrupts execution by pushing the "programmer switch" interrupt button on the left side of the Macintosh (the rearmost of the programmer switch pair);
- 2) When most of the fatal system errors occur; or
- 3) When the *Debugger* trap (`$A9FF`) or *DebugStr* ('aString') trap (`$ABFF`) is executed.

Lightspeed Pascal is distributed with two members of the Macsbug family: MaxBug, which uses a full-size alternate screen, and knows about procedure and trap names; and MacXLBug, which works with the Mac XL (a Lisa with MacWorks) and also uses a full-size alternate screen.

To install Macsbug on a disk, use the Finder to move the appropriate debugger onto your boot disk. Rename the debugger file as *Macsbug*, then choose **Shut Down** from the **Special** menu. Re-insert your disk, and the Macintosh will boot. The boot screen will then say **Macsbug installed** in addition to **Welcome to Macintosh** (unless you have a custom start up screen).

The Macintosh expects the debugger to be named *Macsbug*, so if you don't want Macsbug installed when you boot, you can just rename the file something other than *Macsbug*.

Using Macsbug

Macsbug will take control when you hit the interrupt switch, when most fatal system errors occur or when a *Debugger* or *DebugStr* trap is executed. When a system error occurs, the error is displayed and you receive a `>` prompt. (There are some system errors which will still give you the "bomb" dialog box. Most of the time you can hit the interrupt button to get into Macsbug

from there.) On other breaks, Macsbug will display the machine state (the program counter (PC)), the current instruction, the status register (SR), the data registers (D0 through D7), the address registers (A0 through A7) and prompt you for a command with a `>`.

Command Syntax

The following sections summarize the Macsbug command syntax.

Program Execution Control

<i>G</i>	<i>a</i>	<u>G</u> o; resume program execution at address <i>a</i> (if <i>a</i> is omitted, the program resumes at the value of the PC)
<i>T</i>		<u>T</u> race; execute one instruction (treats A-Traps as one instruction)
<i>S</i>	<i>n</i>	<u>S</u> tep; execute <i>n</i> instructions (<i>n</i> defaults to 1), step into A-Traps
<i>BR</i>	<i>a</i> <i>n</i>	<u>B</u> reakpoint; interrupt program execution the <i>n</i> -th time the PC reaches <i>a</i> (<i>n</i> defaults to 1). Up to six breakpoints may be active at a time.
<i>CL</i>	<i>a</i>	<u>C</u> lear the breakpoint at address <i>a</i> . If <i>a</i> is omitted, <i>CL</i> clears all breakpoints.
<i>GT</i>	<i>a</i>	<u>G</u> o <u>T</u> ill; interrupt program execution the next time the PC reaches <i>a</i>
<i>ST</i>	<i>a</i>	<u>S</u> tep <u>T</u> ill; step through instructions until the PC reaches <i>a</i>
<i>MR</i>		<u>M</u> agic <u>R</u> eturn. Execute until the PC reaches the address on the top of the stack. Most useful when issued just following a <i>JSR</i> or <i>BSR</i> , to return to the instruction after the call.

Note: *BR*, *GT*, and *MR* will not work if the target address is in ROM. To break in ROM you must use *ST*.

Set and Display Commands

<i>DM</i>	<i>a</i> <i>n</i>	<u>D</u> isplay <u>M</u> emory; starting at address <i>a</i> , display <i>n</i> bytes (<i>n</i> defaults to \$10 (hex 10)). <i>n</i> may alternatively take one of three four-character string values, and the data is displayed accordingly. These possible values for <i>n</i> are: ' <i>IOPB</i> ' - display as I/O parameter block ' <i>WIND</i> ' - display as window record ' <i>TERC</i> ' - display as text edit record
<i>SM</i>	<i>a</i> <i>v</i> <i>v</i> <i>v</i> ...	<u>S</u> et <u>M</u> emory; starting at address <i>a</i> , replace memory by values <i>v</i>

<i>Dn v</i>	<u>D</u> ata <u>R</u> egister; sets data register <i>n</i> to value <i>v</i> (if <i>v</i> is omitted, displays the data register value)
<i>An v</i>	<u>A</u> ddress <u>R</u> egister; sets address register <i>n</i> to value <i>v</i> (if <i>v</i> is omitted, displays the value in the address register)
<i>PC v</i>	<u>P</u> rogram <u>C</u> ounter; sets the PC to value <i>v</i> (if <i>v</i> is omitted, displays the value of the PC)
<i>SR v</i>	<u>S</u> tatus <u>R</u> egister; sets the SR to value <i>v</i> (if <i>v</i> is omitted, displays the value of the SR)
<i>TD</i>	<u>T</u> otal <u>D</u> isplay; displays the next instruction to execute, all the registers, the PC and SR
<i>IL a n</i>	<u>I</u> nstruction <u>L</u> ist; starting at address <i>a</i> , disassemble and display <i>n</i> instructions (<i>n</i> defaults to \$10)
<i>ID a</i>	<u>I</u> nstruction <u>D</u> isassembly; disassemble and display the instruction at address <i>a</i>

Expressions

Wherever a command accepts an address or a value an expression may be substituted. The following notations are used in Macsbug expressions:

+	Addition
-	Subtraction
.	Last address given to <i>DM</i> , <i>SM</i> , <i>IL</i> or <i>ID</i>
<i>RAn</i>	The value in address register <i>n</i>
<i>RDn</i>	The value in data register <i>n</i>
@	"Contents of" operator
\$	Interpret the following integer as hexadecimal (default)
&	Interpret the following integer as decimal
<i>PC</i>	the value of the PC
'xxx...'	the value of the given string

Macsbug understands trap names. A trap name evaluates to its trap number. In addition, if you are examining code that was compiled with the N (Names) compile option on, you may use your Lightspeed Pascal function names in expressions. For more details, see Chapter 7, "Debugging", and Chapter 12, "Compile Options" in the User's Guide.

A-Trap Commands

These commands monitor the 68000 *A-Trap* instructions, which are used to invoke Macintosh Operating System and Toolbox routines in the ROM. These commands take up to 6 parameters. *T1* and *T2* indicate the range of A-Traps to monitor, *A1* and *A2* specify the address range which to monitor for A-Trap calls, and *D1* and *D2* specify the range in which D0 must be for the break to occur. All of these parameters are optional. If both *T1* and *T2* are absent, *T1* defaults to 0, *T2* to \$1FE. If *T1* is the only argument, the command is only active for that trap. *A1* defaults to

0, *A2* to the top of memory, *D1* to 0 and *D2* to *\$FFFFFFFF*.

In using the A-Trap commands, the trap numbers *T1* and *T2* are actually the low order 9 bits of the trap instruction. For instance, to break on the trap *\$A97D* (*NewDialog*), use the value *\$17D*. In addition, you may use the trap name instead of the number.

<i>AB</i>	<i>T1</i>	<i>T2</i>	<i>A1</i>	<i>A2</i>	<i>D1</i>	<i>D2</i>	causes a break when the conditions are met
<i>AT</i>	<i>T1</i>	<i>T2</i>	<i>A1</i>	<i>A2</i>	<i>D1</i>	<i>D2</i>	traces the A-Traps meeting the conditions
<i>AH</i>	<i>T1</i>	<i>T2</i>	<i>A1</i>	<i>A2</i>	<i>D1</i>	<i>D2</i>	does an <i>HC</i> (see below) before executing the trap when conditions are met. <i>T1</i> must be at least <i>\$2F</i>
<i>AR</i>	<i>T1</i>	<i>T2</i>	<i>A1</i>	<i>A2</i>	<i>D1</i>	<i>D2</i>	remembers the last A-Trap which met the given conditions. Use <i>AR</i> without arguments to list the trap remembered
<i>AX</i>							clears all A-Trap commands

Program Termination

<i>RB</i>	<u>R</u> e <u>B</u> oots the Macintosh (This doesn't always work well—especially on HyperDrive—use the reset button)
<i>ES</i>	<u>E</u> xit to <u>S</u> hell: returns to the Finder (or to Lightspeed Pascal)
<i>EA</i>	<u>E</u> xit to <u>A</u> pplication: relaunches the current application

Heap Zone Commands

<i>HD mask</i>	<u>H</u> ea <u>p</u> <u>D</u> ump: Displays a list of all blocks in the current heap. The display format is:
----------------	--

addr type size [flags pointer] [] [refnum id type]*

mask is optional. It limits the heap display and summary to those blocks which satisfy the condition in the mask. Possible values for the mask are:

- 'H': only "handle" (relocatable) blocks are displayed
- 'P': only "pointer" (non-relocatable) blocks are displayed
- 'F': only free blocks are displayed
- 'R': only resource blocks are displayed
- 'xxxx': blocks of the given resource type are displayed

HT mask Heap Total. Displays a heap summary line for the *mask* given. The format of the summary line is:

*HLP PF #handle-blocks #locked-blocks #purgeable-blocks
space-occupied-by- purgeable-blocks #pointer-blocks free-space*

HC Heap Check. Checks the consistency of the heap.

Warning: This command will hang if the heap is
munged badly.

HX Heap eXchange. Toggles the heap display between system and application heaps.

Miscellaneous Commands

F a n d m Find. Starting at address *a*, search *n* bytes for data *d*, after masking the data with *m*.

WH a WHere. If *a* is less than 512, give the PC at which the trap code resides. Otherwise, display the trap and trap starting address which is closest to *a*.

CV a ConVert. Displays *a* in unsigned hexadecimal, signed hexadecimal, signed decimal and text. When used with expressions, you have an all-purpose programmer's calculator!

CS a1 a2 CheckSum. Does a checksum of the address range *a1* . . *a2*. When no arguments are present, it returns *CHKSUM T* if the checksum of the last given range is the same as when it was last calculated, *CHKSUM F* otherwise.

SS a1 a2 Step Spy. Does a checksum of the contents of the address range *a1* . . *a2* between the execution of each instruction. Breaks when the checksum changes.

Warning: Very slow but very useful. Disk
access does not work well with
SS active.

AS A1 A2 A-Trap Spy. Similar to Step Spy, but calculates the checksum before each A-Trap. (This command is cancelled by *AX*.)

SC Stack Crawl. This assumes that *LINK* and *UNLK* instructions have been performed at the beginning and end of each function (generally true in Lightspeed Pascal). It returns a list of the active stack frames and their return addresses. This is useful to see who called what and where. (Hint: If you are at a *LINK* instruction, step beyond it before you do *SC*.)

<i>PX</i>	<u>P</u> rocedure symbol <u>eX</u> change. Toggles the visibility of procedure names in <i>IL</i> and <i>ID</i> commands. Defaults to enabled.
<i>RX</i>	<u>R</u> egister <u>eX</u> change. Toggles the visibility of the register dump during tracing and stepping. Defaults to enabled.
<i>DX</i>	<u>D</u> ebugger <u>eX</u> change. Toggles Debugger trap entry. When enabled, calling <i>Debugger()</i> or <i>DebugStr()</i> will break into Macsbug; otherwise it won't cause a break. Defaults to enabled.

Handy Hints

Depressing the return key will repeat the previous command. Exceptions: after an *IL* or *ID* command, hitting return will display the next instructions; after *DM* it will display the next memory locations. After an *MR* command, it will execute a *T* (Trace).

Hitting the backspace key while a Macsbug command is listing will terminate the command. Depressing the space key will pause the listing. Hitting the space key again will restart the display.

Hitting the ` key (at the top left of the keyboard) while in Macsbug will toggle the display between the Macsbug screen and the program's screen.

Sometimes you will break into Macsbug with the disk still spinning. To stop the disk, enter *DM DFF1FF*. This will hit the Woz machine (disk drive controller) address that stops the disk.

The word *\$4E71* is the *NOP* instruction. If you want to take instructions out of your program, use this word to *NOP* out instructions in your code stream.

Note: The *NOP* is a two byte instruction. If the instruction you are going to *NOP* is longer than two bytes, remember to use enough *NOPs* to clear out the whole instruction.

Appendix H

RMaker Reference

Resource Files and Macintosh Program Design	H-3
Using RMaker	H-3
RMaker File Format	H-3
Individual Resources and Their RMaker Constructs	H-5
'ALRT' - Alert Template	H-5
'BNDL' - Bundle Template for Application	H-6
'CNTL' - Control Template	H-6
'DITL' - Dialog (or Alert) Item List	H-6
'DLOG' - Dialog Template	H-7
'FREF' - File Reference	H-7
'GNRL' - General	H-8
'MENU' - Menu	H-8
'PROC' - Procedure (contains code)	H-8
'STR' - String	H-8
'STR#' - String List	H-9
'WIND' - Window	H-9

Appendix H

RMaker Reference

Resource Files and Macintosh Program Design

Macintosh programs are designed around *objects*. Windows, menus, dialog boxes, and alerts are all objects which the Macintosh knows about. These objects may be constructed "on the fly" from within a Macintosh program, or they may be specified independently from the program and stored in the resource fork of the application's file. This independent specification has the advantage of separating the program's functions (as specified in the code) from the program's user interface design (its "look"). This is an important first principle in designing programs for the Macintosh.

RMaker is a *resource compiler*. It takes a textual specification of the objects to be used in a program and produces the resource data structures which are understood by the Macintosh Toolbox routines.

Using RMaker

To create a resource file, use any standard text editor (such as the Text Editor demo provided with Lightspeed Pascal) to create the RMaker source file. Transfer to RMaker. It will display a file selection box and wait for you to choose a file. Choose your RMaker source file as input, and RMaker will produce a resource file from the given source.

RMaker File Format

RMaker input is line-oriented and has a quite rigid syntax. The RMaker input file consists of an output specification followed by an arbitrary number of resource definitions separated by blank lines. Comments are also allowed, either as whole lines or as tags at the end of lines. Comment lines begin with an asterisk *. Comments at the end of lines are preceded by two semicolons ; ;

RMaker files begin with the output specification: the name of the output file on one line, followed by a line assigning a file signature. The file signature is a four-character file type followed by a four-character file type. For example, if you want to name the output file *Sample* and give it the file type *????* and creator *SAMP*, the first two lines of the RMaker input file would be:

```

Sample           ;; the name of the output file
????SAMP        ;; the file type ???? and creator SAMP

```

The file signature line may be left blank, in which case the file type and file creator will be set to nulls (four bytes of 0 each). The output specification may be preceded by an arbitrary number of blank lines or comment lines.

If the file name line starts with an exclamation point, this tells RMaker to merge the resources in the RMaker specification into the file name given. In this case, omitting the file signature will leave the file's signature unchanged.

The body of an RMaker source file consists of declarations of groups of resources headed by a resource type clause. Resource type sections are introduced by a *TYPE* declaration:

Note: In the following syntax, the brackets *[]* enclose optional data. Words in italics describe user-supplied data.)

```

TYPE resource type [= resource type ]

```

If the optional "=" clause is not present, the first resource type must be a predefined type. If the clause is present, the resource type named there must be a predefined type. The = clause allows users to define their own resource types.

RMaker resource declarations are always in the following form:

```

[name], ID [(attributes)]
type-specific resource data

```

The attributes byte must be surrounded in parentheses. See the "Resource Manager" chapter of *Inside Macintosh* for the definition of resource attributes.

The statement

```

INCLUDE filename

```

tells RMaker to take the resources from *filename* and include them in this resource specification.

Numbers are decimal by default.

Special characters may be entered with a backslash followed by two hex digits giving the ASCII code of the character (e.g. `\14` for the apple symbol).

Put `++` at the end of a line to signify that the line is continued on the next line.

Individual Resources and Their RMaker Constructs

Resource definitions are grouped according to type. A resource type group is headed by the word *TYPE* followed by the resource type name on one line, followed by definitions of resources of that type. Resource definitions consist of a first line containing the resource name (optional), followed by a comma and a number, followed by a resource attribute byte in parentheses (optional). Following this first line is resource-specific data which may be on one or many lines, according to the type of the resource.

Resource definitions must be terminated by a blank line.

There are twelve defined resource types recognized by RMaker:

- 'ALRT' - Alert
- 'BNDL' - Bundle
- 'CNTL' - Control
- 'DITL' - Dialog (or Alert) Item List
- 'DLOG' - Dialog
- 'FREF' - File Reference
- 'GNRL' - General
- 'MENU' - Menu
- 'PROC' - Procedure (contains code)
- 'STR' - String
- 'STR#' - String List
- 'WIND' - Window

The following sections illustrate the different types of resource declarations:

'ALRT' - Alert Template

```
TYPE  ALRT
    , 128      ;; the resource number
70 100 150 412 ;; rectangle for the alert (top left bottom right)
10          ;; the resource ID for the item list
FFFF       ;; stages word (in hex)
```

* always remember the blank line at the end of a resource
* definition - it is a required separator

'BNDL' - Bundle Template for Application

```
TYPE  BNDL
      ,128      ;; resource number
SAMP 0          ;; creator for bundle
ICN#           ;; resource type (icon list)
0 128 1 129    ;; local ID 0 maps to ICN# resource 128, 1 to 129
FREF          ;; resource type (file reference)
0 128 1 129    ;; local to FREF mapping 0 to 128, 1 to 129
```

'CNTL' - Control Template

```
TYPE  CNTL
      ,128      ;; resource number
MyControl      ;; title for control
10 10 20 20    ;; rectangle for control (top left bottom right)
Visible        ;; may also be Invisible
0              ;; CDEF proc ID
0              ;; reference constant (defines control type)
0 100 0        ;; minimum value, maximum value, initial value
```

'DITL' - Dialog (or Alert) Item List

```
TYPE  DITL
      ,10      ;; resource number
9      ;; number of items in the item list

button      ;; enabled button items (enabled by default)
20 20 40 100 ;; rectangle (window-relative coordinates)
Cancel      ;; text in the button

radioButton ;; radio button item
50 20 70 120 ;; rectangle (includes button and text)
Push Me     ;; the text will go to the right of the button

radioButton disabled ;; dimmed radio button item
50 20 70 120 ;; rectangle (includes button and text)
Can't Push Me ;; you can't push a disabled button

checkBox    ;; check box item
80 20 100 120 ;; rectangle (includes check box and text)
Check Me    ;; the text goes to the right of the box
```

```

staticText      ;; static text item
20 120 40 320   ;; the rectangle into which the text is placed
This text would get placed next to the Cancel button ;; the text

editText Disabled ;; disabled editable text item
50 140 90 320   ;; coordinates of the box for editable test
initial string  ;; the edit test gets initialized to this string.

editText      ;; editable text item (enabled by default)
100 140 120 320 ;; rectangle
you can edit this text ;; the initial string for editable item

iconItem      ;; for the display of an icon
100 100 132 132 ;; rectangle should be 32x32
3             ;; resource ID for icon (Type ICON)

picItem       ;; for the display of a Quickdraw picture
30 100 20 200  ;; display rectangle (picture will be scaled)
57            ;; resource ID for picture (Type PICT)

userItem      ;; a user-defined item
30 40 80 90   ;; the rectangle

```

'DLOG' - Dialog Template

```

TYPE DLOG
    ,128      ;; the resource number
My Dialog Box ;; a message
70 100 150 412 ;; the rectangle (top left bottom right)
Visible NoGoAway ;; may also be Invisible, and may also be GoAway
0             ;; the dialog definition ID
0             ;; the refCon, available to the user
10           ;; the resource ID for the dialog item list

```

'FREF' - File Reference

```

TYPE FREF
    ,128      ;; the resource number
APPL 0       ;; the file type and local ID

```

'GNRL' - General

'GNRL' is used to define your own resource types and define their format. The resource's format is constructed from "elements". The elements available are:

- .P Pascal string
- .S String without a leading length byte
- .I Decimal integer
- .L Decimal 32-bit integer
- .H Hexadecimal integer
- .R Read the given resource from the given file.
 .R takes the arguments *filename resource type resource ID*

```
TYPE ICN# = GNRL      ;; define the type ICN#
    ,128              ;; the resource ID
.H                    ;; hexadecimal data follows
0001 8000 0002 4000    ;; ICN#'s need 2 icons (icon and
0003 C000 0004 2000    ;; mask) of 32x32 bits each, or 32
...                    ;; lines of two longwords apiece.
FFFF FFFF FFFF FFFF
```

'MENU' - Menu

```
TYPE MENU
    ,10                ;; the resource number (Menu ID)
MyMenu                ;; the menu title
First Item            ;; the first menu item
Second Item /S        ;; the second menu item with a command-key "S"
(Third Item           ;; the third item (disabled by the preceding "(")
(-                    ;; a grey line (this is item #4)
Fifth Item            ;; the fifth item
```

'PROC' - Procedure (contains code)

```
TYPE PROC
    ,128                ;; the resource number
Filename              ;; the code from this file will get placed
                      ;; in the resource
```

'STR' - String

```
TYPE STR              ;; spelled 'STR' - trailing space required!
    ,128                ;; the resource number
My Wild Irish Rose    ;; the string assigned to the resource 128
```

'STR#' - String List

```
TYPE  STR#  
    ,128          ;; the resource number  
    2             ;; the number of strings in the list  
    The First String  
    And the second string    ;; the two strings in the list
```

'WIND' - Window

```
TYPE  WIND  
    ,128          ;; the resource ID  
    My Window     ;; the window title  
    40 40 200 472  ;; the window rectangle (top left bottom right)  
    Visible GoAway ;; may also be Invisible and may also be NoGoAway  
    0             ;; the window definition ID  
    0             ;; refCon, a long word available to the User
```


Appendix I

ResEdit Reference

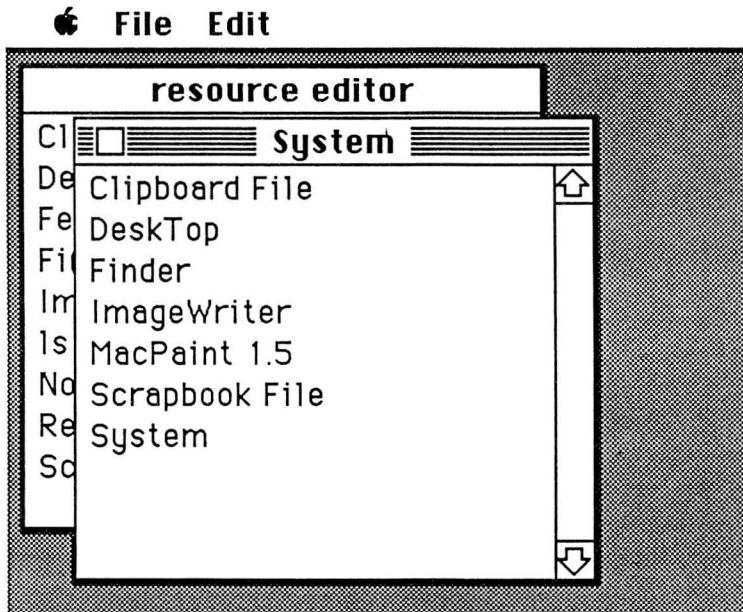
Introduction	I-3
The CURS Editor	I-5
The DITL Editor	I-6
The FONT Editor	I-6
The ICN# Editor	I-7
The WIND and DLOG Editor	I-8
Other Resource Type Editors	I-8
The General Editor	I-10

ResEdit Reference

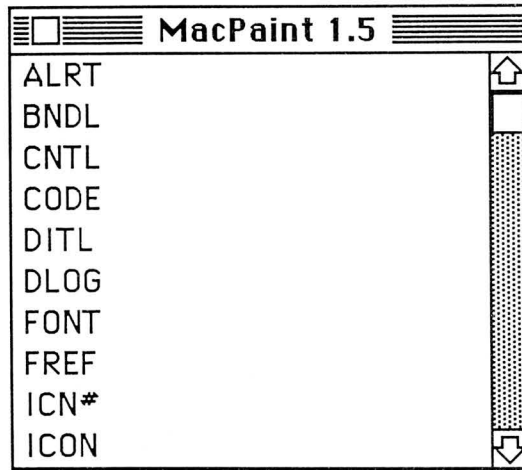
Introduction

ResEdit is a graphically based resource editor, which allows the creation of resources and editing of existing resources. It is composed of several individual editors for specific resource types, plus a general editing facility which lets you edit the hexadecimal resource data directly for any resource type.

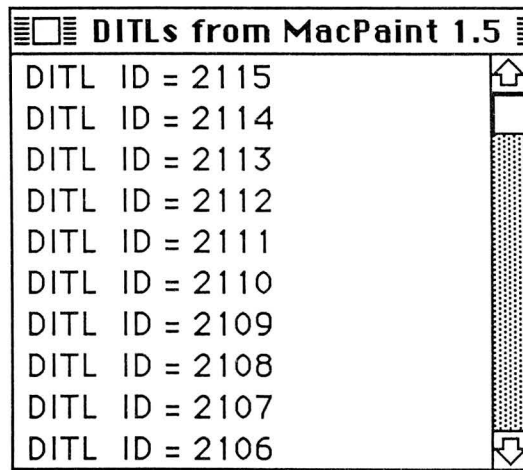
When you launch ResEdit, you get a window for each of your mounted disks listing the files on the disk.



Opening a file shows a list of all the resource types in the resource fork of the file.



If there is no resource fork for the file, ResEdit will inquire whether you want to create a resource fork for the file. Opening a resource type lists the individual resources of that type in the file.



Opening a particular resource will invoke the specific resource editor for that type if there is one. If not, the general (hexadecimal) resource data editor is invoked.

ResEdit is easy to use and largely self-documenting. Essentially, what you see is what you get. All of the resource editors either put up dialog boxes that you fill in, or provide pictures of the resources that you manipulate graphically.

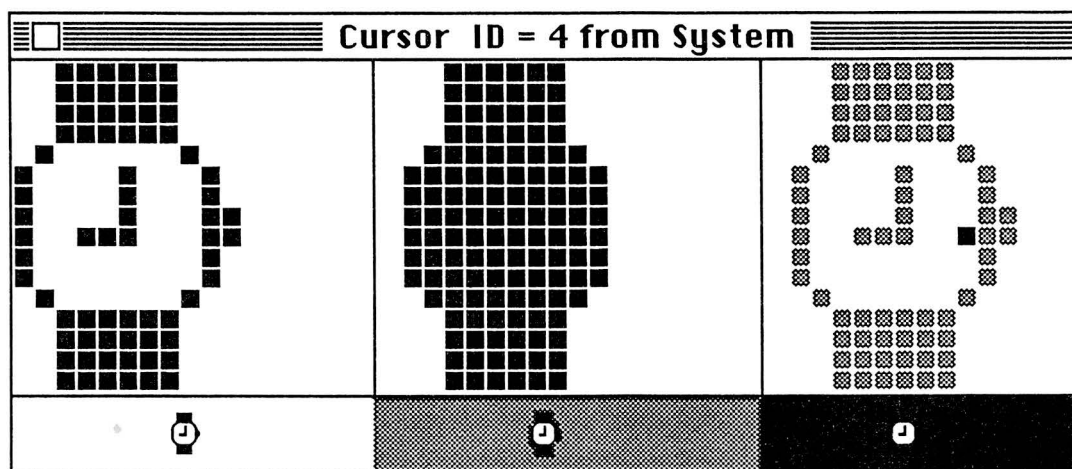
The standard meanings of the **File** and **Edit** menus are altered slightly depending upon what kind of window is in front. For instance, **New** will create a new instance of the kind of element in the front window. If the disk contents window is in front, it creates a new file; if the file resource type list is in front, a new resource type is created; if a resource ID list is in front, a new resource of that type is created. The **Edit** menu commands similarly operate on the type of window that is in front.

When a particular resource is selected out of a resource ID list, or whenever a window for a particular resource is front, the command **Get Info** will put up a dialog box containing that resource's parameters in editable fields. You can use this feature to change the resource ID, set resource attributes and other properties particular to that resource.

When you double-click on a particular resource, the resource editor for that type of resource is invoked. If no specialized editor exists, the general resource editor is invoked.

The CURS Editor

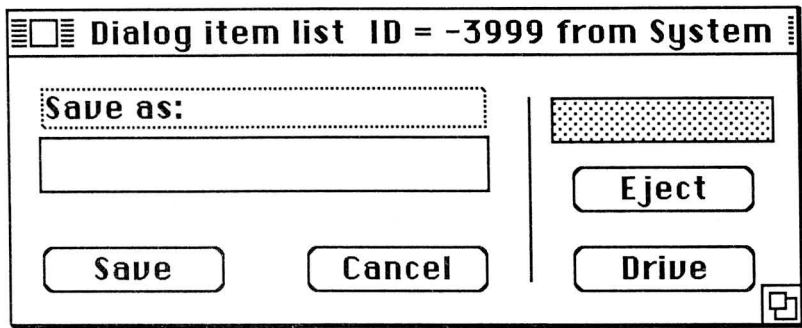
When the *CURS* editor is invoked, it displays a window with three images of the cursor which may manipulated by the mouse.



The leftmost image is how the cursor will appear. The image in the center is the mask for the cursor. This will determine how the cursor will appear on different backgrounds (as you can see on the display beneath the editable cursor fields). The rightmost panel lets you set the cursor's hotspot, which appears at a black dot among the gray dots of the cursor.

The DITL Editor

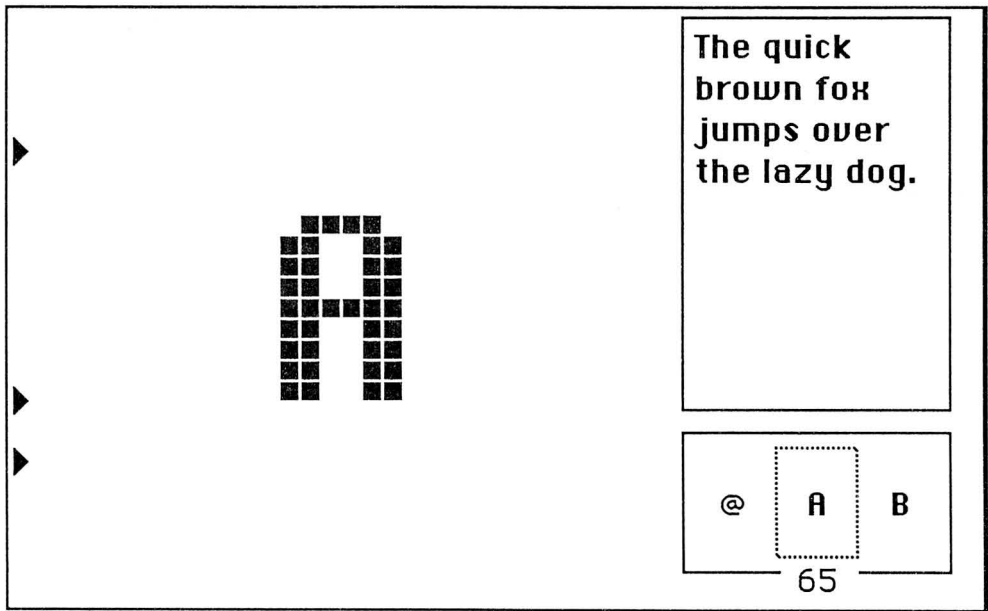
When you start the *DITL* resource editor, a window is displayed showing the dialog items as you would see them in the dialog window.



You can select items by double-clicking on them, which brings up a window allowing you to edit that item individually. You can change the size of an item by clicking and dragging the lower right corner of the item. You can move an item by dragging it with the mouse. When the dialog window is front, the menu command **New** will create a new dialog item.

The FONT Editor

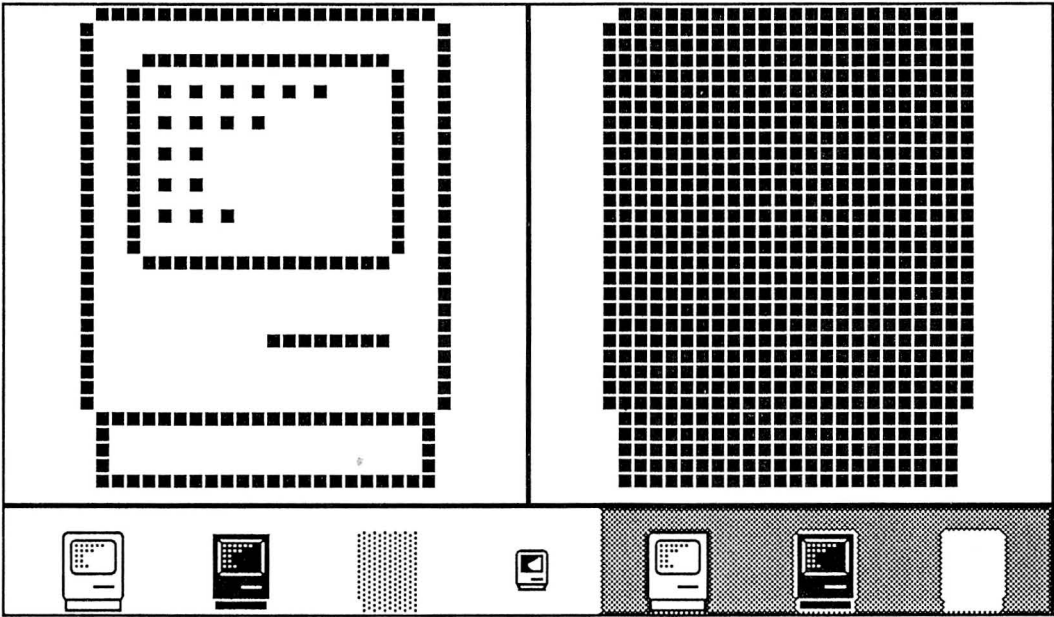
The *FONT* editor displays three panes: the sample text pane, the character selection pane and an editable window which shows one character at a time.



The selection pane shows three characters. Clicking on the left character in the pane moves you downwards in the character set, clicking on the right character moves upward through the set. You select a character in the selection pane, and edit the character in the edit pane. The three triangles in the edit pane represent the font ascent, baseline and descent.

The ICN# Editor

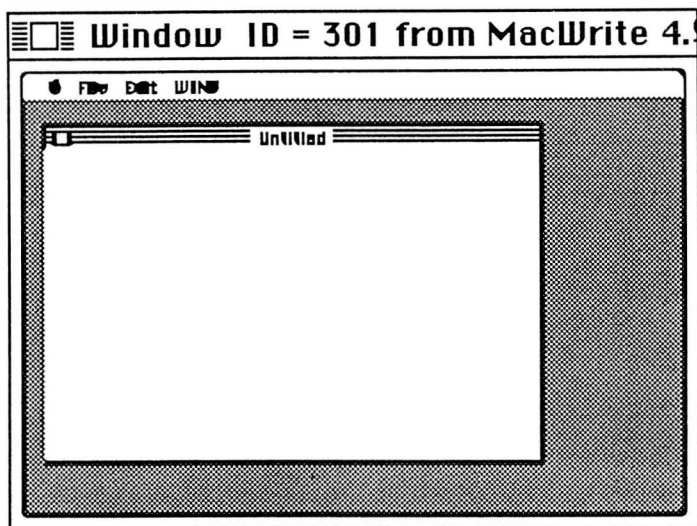
The *ICN#* editor displays two panes:



The left pane represents the icon, the right the icon mask. Along the bottom of the window the icon is displayed in its unselected, selected and dimmed states on white and gray backgrounds. You use the mouse to set bits in the large icon and mask panes to create your icons.

The WIND and DLOG Editor

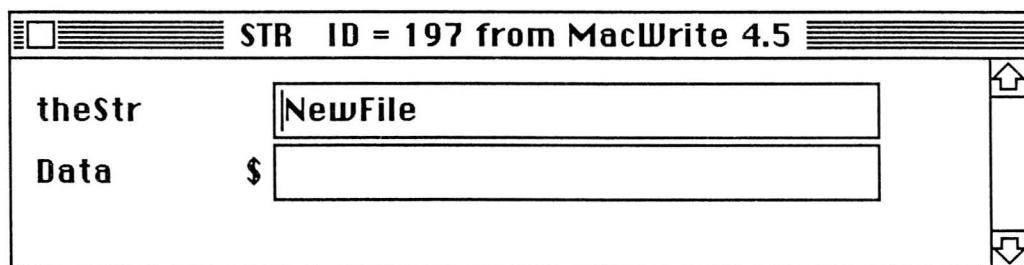
The *WIND* editor displays a window with a picture of a window on a desktop.



You can grow the window picture by dragging its lower right corner. You move this window picture by dragging its title bar. This editor is also invoked for the resource type *DLOG*.

Other Resource Type Editors

ResEdit allows you to edit other types of resources with a dialog box style editor. The fields of the particular resource type may be set by typing the information for the field into the box along side the field name. Here's an example of the dialog box you get for the type *STR*:



Here's what you get for the resource type *MENU*:

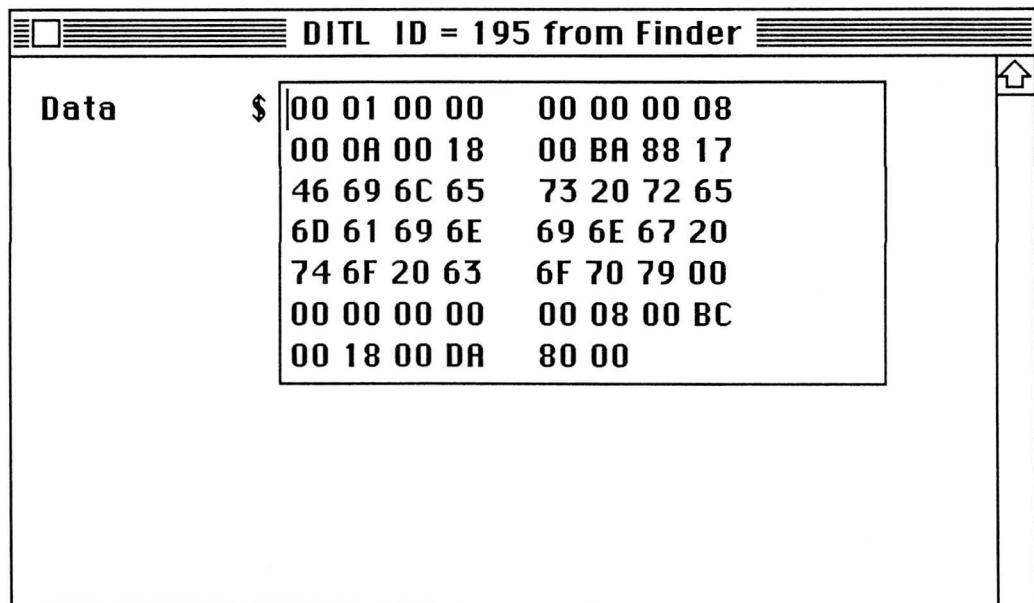
MENU ID = 4 from MacWrite 4.5	
menuID	4
width	0
height	0
procID	0
filler	0
enableFlgs	\$FFFFFFFF
title	Search

menuItem	Find...

Click on the row of asterisks to select the menu item below the row. Double-click on the row of asterisks to insert a new menu item below the row.

The General Editor

The general editor shows a dialog box with one editable field labeled **Data**.



You may change the hexadecimal data in this field. Make sure you understand the data format for the resource before editing it! (Resource format documentation can be found in *Inside Macintosh*.)

Appendix J

Macintosh Character Set

LSD \ MSD																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	SP	0	@	P	`	p	Ä	ê	†	∞	ı	—	‡	🍏
1	SOH	DC1	!	1	A	Q	a	q	Å	ë	°	±	ı	—	.	Ò
2	STX	DC2	"	2	B	R	b	r	Ç	í	¢	≤	¬	“	,	Ú
3	EXT Enter	DC3	#	3	C	S	c	s	É	ì	£	≥	√	”	„	Û
4	EOT	DC4	\$	4	D	T	d	t	Ñ	î	§	¥	f	‘	‰	Ü
5	ENQ	NAK	%	5	E	U	e	u	Ö	ï	•	μ	≈	,	Â	ı
6	ACK	SYN	&	6	F	V	f	v	Ü	ñ	¶	∂	Δ	÷	Ê	^
7	BEL	ETB	'	7	G	W	g	w	á	ó	ß	Σ	«	◇	Á	~
8	BS	CAN	(8	H	X	h	x	à	ò	®	Π	»	ÿ	Ë	-
9	HT	EM)	9	I	Y	i	y	â	ô	©	π	...	ÿ	È	˘
A	LF	SUB	*	:	J	Z	j	z	ä	ö	™	∫		/	Í	·
B	VT	ESC Clear	+	;	K	[k	}	ã	õ	´	¸	À	⊠	Î	°
C	FF	FS	,	<	L	\	l		å	ú	”	º	Ã	<	Ï	,
D	CR	GS	-	=	M]	m	}	ç	ù	≠	Ω	Õ	>	Ì	”
E	SO	RS	.	>	N	^	n	~	é	û	Æ	æ	Œ	fi	Ó	˘
F	SI	US	/	?	O	_	o	DEL	è	ü	Ø	ø	œ	fl	Ô	˘

Row and column headings are hexadecimal digits.

Columns are the most significant digit, rows are the least significant digit.

(Column*16) + Row gives the numeric code for the character.

The first 32 characters (00-1F) and DEL (7F) are nonprinting control codes.

The other characters are shown in Times 18 point font. Different fonts may produce different characters.

In Chicago font, the 🍏 character is the control code 14.

Index

Page numbers in this index are listed using the same conventions as the actual page numbering in the manual:

1. Page numbers in the *User's Guide* contain both the chapter number and the page number. For example, 1-1 refers to Chapter 1, page 1 of the *User's Guide*.
2. Page numbers in the *Language Reference* are sequential and lack section numbers. For example, 15 refers to page 15 of the *Language Reference*.
3. Page numbers in the *Appendices* contain both the appendix letter and the page number. For example, H-1 refers to page 1 of Appendix H.

A range of consecutive pages is indicated by the symbol .. between two page numbers.

- @ 53..54
 - in ANS Pascal A-3
- [and]
 - in **array** type 23
 - in **string** type 27
 - in **array** index 37
 - in set constructor 55
- - in real number 5
 - in field reference 38
- - in **with** statement 70
 - in **inline** declaration 77
 - in **case** statement 65
 - in function call 54
 - in **label** declaration 11
 - in enumerated type 20
 - in **array** type 23
 - in **record** type 25
 - in array index 37
- (and)
 - in program heading 89
 - in enumerated type 20
 - in record type 25
 - in expressions 44
 - in function call 54
 - in type casts 56
 - in parameter lists 80
- :
 - in record type 24
 - in variable declaration 35
- after label in statement 59
- in **function** declaration 78
- in parameter list 81
- ;
 - in unit 90
 - in program 89
 - in record type 24, 25
 - in variable declaration 35
 - in **case** statement 45
 - in **procedure** declaration 75
 - in **function** declaration 78
 - in parameter list 80
- ^
 - in pointer type 28
 - in qualifier 36
 - in file buffer reference 38
 - in pointer reference 39
- { and }
 - in comments 7
- \$
 - in hexadecimal integers 5
 - in compile options 5-4, 7
- ..
 - in subrange type 21
 - in set constructor 55
- (* and *)
 - in comments 7
- :=
 - in assignment statement 57
 - in **for** statement 68

3-D Graphics (see Three-Dimensional Graphics)

A

A5 world 11-2
About Lightspeed Pascal 15-1
abs function 123
accessing files 96
accessing global symbols 11-7
accounting applications D-5..D-6
activating
 a block 13
 a window 4-3
Add File... command 3-3, 5-2, 12-10,
 15-6..15-7
Add Window command 5-2, 15-6..15-7
AddPt procedure in QuickDraw C-44
allocating space for QuickDraw regions C-11
and operator 43, 45, 49
anonymous file 96
ANS Pascal A-3
 exceptions to requirements A-3
 extensions A-4
 standard errors A-6
Apple Menu 15-1
Apple utilities (see utilities)
AppleTalk Manager interface E-1
application data types D-5, D-33
applications
 building 9-1
 compile options in 9-5
 using resources with 10-1
arctan function 126
arithmetic
 arithmetic type D-5, D-33
 in SANE D-9..D-10
 operators 19, 47
 with real type values 21
array
 array type 23
 component type 23
 index 37
 index type 23
assembly language
 interfacing with 11-1, 11-7..11-8
assignment compatibility 28..30
assignment statement 59..60
A-Traps
 Breaking 14-4
 handlers (bottlenecks) 14-4
 in Macsbug commands G-5
 modified 14-5..14-6

Auto-Save command 6-4, 15-8..15-9
Auto-Show Finger command 7-4, 15-10

B

BackColor procedure in QuickDraw C-43
BackPat procedure in QuickDraw C-36
base type 26, 28
baseAddr field in a bitmap C-14
begin in compound statement 63
binary
 conversion to decimal D-12
 floating point number D-33
 operators 43, 47
 scale functions D-26
 table of arithmetic operations 47
Binary-Decimal Conversion Package interface
 E-7
BInlineF procedure 136
bit image C-12
BitAnd function 132
bitmap data structure C-13
BitNot function 132
BitOr function 132
BitXor function 132
bkColor field in **grafPort** C-19
bkPat field in **grafPort** C-19
blank separators 3
block 11
 activation 12..13
 definition of 11
bold field in **grafPort** C-22
boolean
 operators 49
 result of operation 49
 type 19..20
bounds field in a bitmap C-14
Break at A-Traps command 12-9, 15-10..15-11
Bug Spray Can 3-14..3-15, 6-2, 6-4, 7-3
Build & Save As... command 3-17, 7-2, 9-1,
 9-3, 9-5, 10-2, 13-3, 15-6..15-7
Build command 2-2, 6-4, 15-8
build order 5-3, 6-2
building
 applications 9-1
 libraries 9-1
 without running (see **Build** command)

C

calling conventions
 calling sequence 11-5
 parameter passing 11-6

- return values 11-7
- routine entry 11-5
- routine exit 11-6
- case**
 - in **record** type 25
 - statement 63, 65..66
- cents type 22
- char** type 6, 19
 - with read 104
 - with write 109
- character fonts in QuickDraw C-21
- character pairs 3
- character set 3
 - Macintosh J-1
 - Pascal 3
- character set subset 3
- character string 6
- character strings (see strings)
- CharWidth** function in QuickDraw C-42
- Check** command 6-4, 15-8
- Check Link** command 6-4, 15-8..15-9
- chr** function 126
- class inquiries in SANE D-20
- Clear** command 15-4..15-5
- Clipboard** command 15-11..15-12
- Clipboard** Window 15-11..15-12
- clipping in **grafPort** C-18, C-26
- ClipRect** procedure in QuickDraw C-36
- clipRgn** field in **grafPort** C-18
- close box (see go-away box)
- Close** command 4-4, 15-2
- close** procedure 98
- Close Project** command 2-2, 3-12, 15-2, 15-6
- ClosePicture** procedure in QuickDraw C-62
- ClosePoly** procedure in QuickDraw C-63
- ClosePort** procedure in QuickDraw C-34
- CloseRgn** procedure in QuickDraw C-56
- closing
 - a project 3-12
 - files from editor 4-4
 - files from program 98
- colrBit** field in **grafPort** C-19
- comments 3, 7
- comp** type 22, D-5..D-6, D-33
- comparing types 51..52
- compatibility of types 28..29
- compile directives (see compile options)
- compile options 5-4..5-5, 9-5, 13-1..13-8, 7, 16
 - A (Asynch) 13-8
 - D (Debug) 5-5, 13-2..13-3
 - enabling/disabling 5-5, 13-1..13-2, 13-8
 - I (Initialization) 9-2, 13-8
 - N (Names) 5-5, 13-1, 13-3
 - R (Range Checking) 5-5, 13-1, 13-5..13-7
 - V (Overflow) 5-5, 13-1, 13-4
- compiled code
 - examining 12-9
 - size of 3-3..3-4, 3-6, 5-2
- compiling by build order 6-2
- compiling without running (see **Check** command)
- completion routines 14-4
- component type 23, 26
- compound statements 63
- compressed project 9-5
- computational** type (see **comp** type)
- computational value 22
- concat** function 128
- condense** field in **grafPort** C-23
- conditional statements 63
- Confirm Saves** command 6-4, 15-8..15-9
- const** declaration 7, 11
- constant integer type 18
- constants 3, 7, 11, 13
- context switches for A-Trap Breaking 14-4
- Control Manager interface package E-8
- conversion
 - between binary and decimal D-12
 - from decimal records to decimal strings D-15
 - from decimal records to SANE types D-14
 - from decimal strings to decimal record D-15
 - from decimal strings to SANE types D-12
 - from SANE types to decimal records D-14
 - .Rel files to libraries 11-8
 - to integral formats D-11
- coordinates
 - coordinate plane in QuickDraw C-7
 - of a point in QuickDraw C-8
- Copy** command 7-8, 15-4..15-5
- copy** function 128
- CopyBits** procedure in QuickDraw C-60
- CopyRgn** procedure in QuickDraw C-55
- cos** function 125
- creating
 - a new file 96..98
 - a new project 3-2, 12-10, 15-6
 - grafPort** C-18
 - QuickDraw regions C-11
- current file position 38..39, 96
- CURS** editor in ResEdit I-5
- cursor** fields in QuickDraw C-15..C-16
- customizing QuickDraw operations C-66
- Cut** command 15-4..15-5

D

- data initialization B-6
- data representation B-6
- data structure
 - of a bitmap C-13
 - of a polygon C-31
- data type limitations B-5
- Debug compile option 7-2, 7-4
- Debug Menu
 - Auto-Show Finger command 7-4, 15-10
 - Break at A-Traps command 12-9, 15-10
 - Macsbug command 15-10..15-11
 - Pull Stops command 7-7, 15-10
 - Show Finger command 7-4, 7-7, 15-10
 - Step Into Calls command 7-4..7-5, 15-10
 - Stops In command 3-8, 3-11, 7-5..7-7, 12-11, 15-10
- Debugger procedure 138
- debugging 1-2, 1-4, 3-6..3-18, 7-1..7-10, 16
 - options 5-4 -5-5
 - with LightsBug 12-1..12-15
- DebugStr procedure 138
- decimal
 - conversions to binary D-12
 - conversions to SANE types D-12..D-14
- declaration
 - position 13
 - scope 12
- default bitmap in grafPort C-18
- defining lines in grafPort C-26
- defining polygons in QuickDraw C-32
- Delete... command 15-2, 15-4
- delete procedure 129
- denormalized numbers in SANE D-19, D-33
- Desk Manager interface E-10
- Device and File Manager interfaces E-11
- device field of grafPort C-18
- devices on the Macintosh 115
- Dialog Manager interface E-17
- DiffRgn procedure in QuickDraw C-57
- directives 4
- Disk Driver interface E-20
- Disk Initialization Package interface E-21
- dispose procedure 120
- DisposeRgn procedure in QuickDraw C-55
- DITL editor in ResEdit I-6
- div operator 43, 45, 47..49
- divide by zero exception in SANE D-22
- DLOG editor in ResEdit I-8
- do (see **for**, **while** and **with** statement)
- Don't Save command 6-4, 15-8, 15-10
- double type 22, D-5..D-6, D-33

downto (see **for** statement)

downward rounding direction in SANE D-21

DrawChar procedure in QuickDraw C-42

drawing C-20

in color C-29

in grafPort C-26

Drawing command 15-11..15-12

Drawing Window 6-1, 15-12

DrawLine procedure in QuickDraw C-1

DrawPicture procedure in QuickDraw C-62

DrawString procedure in QuickDraw C-42

DrawText procedure in QuickDraw C-42

dynamic length attribute 27

dynamic variables 28, 35, 39, 119

E

Edit Menu

- Clear command 15-4..15-5
- Copy command 7-8, 15-4..15-5
- Cut command 15-4..15-5
- Everywhere command 4-1, 4-9, 15-4, 15-6
- Find command 4-1, 4-9, 15-4..15-5
- Find What... command 4-1, 4-8, 15-4..15-6
- Paste command 7-8, 15-4..15-5
- Replace command 4-1, 4-9, 15-4..15-5
- Select All command 15-4..15-5
- Show Error command 4-1, 4-7, 15-4..15-5
- Show Selection command 4-1, 4-7, 15-4..15-5
- Undo command 15-4

editing 4-1..4-9

- Add Window command 4-1
- checking syntax errors 4-6..4-7
- closing editing window 4-4
- differences from other Mac editors 4-1
- editing window 4-2
- file and project management 4-1
- indenting feature 4-6
- opening editing window 4-2
- Pascal-specific features 4-1, 4-5..4-7
- pretty printing 4-5
- Project Window 4-1
- Save a Copy As... command 4-1, 4-4
- Save As... command 4-1, 4-4
- Save command 4-4
- search and replace options 4-8..4-9
- selecting text 4-7
- Undo command 4-1

editing window 7-4, 7-6

- activating 4-3
- anatomy of 4-2..4-3
- closing 4-4
- go-away box 4-3

- opening 4-2
- scroll arrows 4-3
- scroll box 4-3
- size box 4-3
- title bar 4-3
- elementary functions in SANE D-26..D-30
- else** (see **if** statement)
- EmptyRect** function in QuickDraw C-49
- EmptyRgn** function in QuickDraw C-58
- end**
 - in **case** statement 65
 - in compound statement 63
 - in **record** type 24
 - in **unit** 90
- entry point to a function 54
- enumerated constant 20
- enumerated types 19..21
 - with read 106
 - with write 111
- environment settings
 - in project 10-2
 - in SANE D-20..D-25, D-33
- eof** function 99
- eoln** function for textfiles 112
- EqualPt** function in QuickDraw C-44
- EqualRect** function in QuickDraw C-49
- EqualRgn** function in QuickDraw C-58
- EraseArc** procedure in QuickDraw C-54
- EraseOval** procedure in QuickDraw C-51
- ErasePoly** procedure in QuickDraw C-64
- EraseRect** procedure in QuickDraw C-50
- EraseRgn** procedure in QuickDraw C-59
- EraseRoundRect** procedure in QuickDraw C-52
- errors 7-1, 6, A-6, D-19, F-1..F-68
- Event Manager
 - OS interface E-24
 - Toolbox interface E-22
- Everywhere** command 4-1, 4-9, 15-4, 15-6
- exceptions
 - exception flags D-21, D-33
 - exception halts D-21
 - exception handlers 14-6..14-7
 - SANE D-22, D-33
 - to ANS Pascal requirements A-3
- execution finger 3-7, 3-11, 7-3..7-4, 7-7
- exp** function 125
- exponent D-33
- exponential functions in SANE D-27..D-28
- expressions 41..56
 - evaluation by SANE D-15..D-17
 - used in Macsbug G-5
- extend** field in **grafPort** C-23

- extended precision D-5
- extended precision expression evaluation in SANE D-16
- extended temporaries in SANE D-15..D-16
- extended** type 22, D-5, D-33
- extensions
 - Lisa Pascal B-10
 - to ANS Pascal A-4
- external declarations 77
- external** directive 8-4..8-5

F

- factors 44
- fan key commands (see equivalent menu command)
- fgColor** field in **grafPort** C-19
- field designator 38
- field list 24
- fields 24
- file buffer 35, 38..39
- File Menu
 - Close** command 4-4, 15-2
 - Delete...** command 15-2, 15-4
 - New** command 4-2, 15-2
 - Open...** command 4-2, 7-10, 15-2
 - Page Setup...** command 15-2..15-3
 - Print...** command 15-2..15-3
 - Quit** command 3-18, 15-2, 15-4
 - Revert** command 15-2..15-3
 - Save a Copy As...** command 4-4, 15-2..15-3
 - Save As...** command 4-4, 7-10, 15-2..15-3
 - Save** command 4-4, 15-2..15-3
 - Transfer...** command 15-2, 15-4
- file** type 26..27, 35
- file variable 96
- file window (see editing window)
- file windows command 15-12
- filepos** function 97, 100
- files
 - closing 4-4
 - file-buffers 38
 - file** type 26
 - in Pascal language 96, 115
 - opening 4-2, 5-2
 - saving without closing 4-4
- FillArc** procedure in QuickDraw C-54
- Filloval** procedure in QuickDraw C-51
- fillPat** field in **grafPort** C-19
- FillPoly** procedure in QuickDraw C-65
- FillRect** procedure in QuickDraw C-50
- FillRgn** procedure in QuickDraw C-59

FillRoundRect procedure in QuickDraw C-53
 financial calculations D-5, D-28..D-29
Find command 4-1, 4-9, 15-4..15-5
Find What... command 4-1, 4-8, 15-4..15-6
Finder 2-1, 2-3
Fixed-Point Math
 interface E-25
 library 8-4
 floating point storage formats in SANE D-6
FONT editor in ResEdit I-6
Font Manager interface E-26
for statement 66..70..70
ForeColor procedure in QuickDraw C-43
 formal parameter lists 80, 85
forward declarations 4, 76
FrameArc procedure in QuickDraw C-53
 framed shapes in **grafPort** C-27
FrameOval procedure in QuickDraw C-50
FramePoly procedure in QuickDraw C-64
FrameRect procedure in QuickDraw C-49
FrameRgn procedure in QuickDraw C-58
FrameRoundRect procedure in QuickDraw C-50
function declaration 12, 78..80
 functional parameters 85
 functions
 abs 123
 arctan 126
 BitAnd 132
 BitNot 132
 BitOr 132
 BitXor 132
 calling 45, 54
 chr 126
 concat 128
 copy 128
 cos 125
 declaring 12, 78..79
 entry point 54
 eof 99
 eoln 112
 exp 125
 filepos 97, 100
 function declaration 12, 78..80
 HiWord 133
 include 130
 input/output 95..116
 length 127
 ln 125
 LoWord 133
 NewFileName 135
 OldFileName 135
 omit 129
 ord 126

ord4 122
pack 121
pointer 123
pos 128
pred 127
 recursive 13, 75..76, 80
round 122
sin 125
sizeof 132
sqr 124
sqrt 124
StringOf 134
succ 127
trunc 122

G

general editor in ResEdit I-10
General-Purpose Data Types interface E-28
Generic LSP disk 2-1..2-2
Generic procedure 137
get procedure 99
GetClip procedure in QuickDraw C-36
GetDrawingRect procedure 131
GetFontInfo procedure in QuickDraw C-43
GetPen procedure in QuickDraw C-38
GetPenState procedure in QuickDraw C-38
GetPixel function in QuickDraw C-65
GetPort procedure in QuickDraw C-34
GetTextRect procedure 131
 global symbols 11-7..11-8
GlobalToLocal procedure in QuickDraw C-45
Go command 3-4, 3-6, 3-9..3-11, 3-13, 3-15..3-17, 6-1, 6-5, 7-3, 7-5, 7-7..7-8, 12-10, 15-8..15-10
 go-away box 4-3
Go-Go command 3-11, 6-1, 7-2..7-3, 7-5, 7-7..7-8, 15-8..15-11
goto statement 61, 61..62
GrafDevice procedure in QuickDraw C-35
grafPort C-17, C33..C-38
grafProcs field in **grafPort** C-20, C-31
 graphics pen C-20

H

heap 11-2

- heap zone commands in Macsbug G-6
- HFS (see Hierarchical File System)
- HideAll procedure 130
- HideCursor procedure in QuickDraw C-37
- HidePen procedure in QuickDraw C-38
- Hierarchical File System 2-3
- HiWord function 133
- horizontal coordinates in QuickDraw C-7
- hotspot in QuickDraw C-16
- how to use this manual 1-4..1-6

I

- ICN# editor in ResEdit I-7
- identical types 28
- identifiers 3..4
 - case of 4
 - length of 4, B-4
- IEEE standard D-11, D-16
- if statement 63..65
- Imagewriter 2-1
- implementation** 90..91, A-4, B-4
- in operator 43, 46, 50, 52
- include function 130
- indexing 37
 - index expression 37
 - index types 23, 37
 - indexed arrays 37
 - multiple expressions 37
- inexact exception in SANE D-23
- infinities in SANE D-18, D-33
- InitCursor procedure in QuickDraw C-37
- InitGraf procedure in QuickDraw C-33
- InitPort procedure in QuickDraw C-34
- inline** declaration 77, 77..78, 136, A-4, B-4
- InlineP procedure 136
- input standard file variable 96
- input/output B-9
 - functions and procedures 95..116
- insert procedure 129
- InsetRect procedure in QuickDraw C-47
- InsetRgn procedure in QuickDraw C-57
- Install command 2-1
- installing
 - Lightspeed Pascal 2-1..2-3
 - Macsbug G-3
- Instant command 15-11..15-12
- Instant Window 3-6, 3-11, 7-3, 7-8..7-10, 15-12
 - Do It button 3-11..3-12, 3-16, 7-8
 - reload with Open... 7-10
 - Save As... 7-10
- integer arithmetic B-7

- integer operands 19
- integer standard identifier 18
- integer type
 - identifier 17..18
 - in SANE D-33
 - with read 104
 - with write 109
- integral value D-33
- interactive I/O B-9
- interface** 90..91, A-4, -B-4
- interface identifiers 13
- interfacing with assembly language 11-1, 11-7..11-8
- International Utilities Package interface E-29
- invalid operation exception in SANE D-22
- InvertArc procedure in QuickDraw C-54
- InvertCircle procedure in QuickDraw C-1
- InvertOval procedure in QuickDraw C-51
- InvertPoly procedure in QuickDraw C-65
- InvertRect procedure in QuickDraw C-50
- InvertRgn procedure in QuickDraw C-59
- InvertRoundRect procedure in QuickDraw C-52
- italic field in grafPort C-22

K

- Kerning C-21
- keywords (see word-symbols)
- KillPicture procedure in QuickDraw C-62
- KillPoly procedure in QuickDraw C-63

L

- label** declaration 11
- labels 4, 6, 11
- lazy I/O 113..115
- length attribute
 - dynamic 27
 - in string types 6
- length function 127
- libraries 5-1, 8-1, 8-4, 11-7
 - building 9-1
 - calling 9-3..9-5
 - compile options in 9-5
 - interface (source) file 8-4
 - library (object code) file 8-4
 - MacPasLib 2-1..2-3
 - MacTraps 2-1..2-3
 - saving projects as 9-3
 - standard Macintosh 8-4

- LightsBug 15-12
 - editing memory display 12-7..12-8
 - entering 12-1
 - example using 12-10..12-15
 - memory display 12-6
 - registers 12-4
 - subroutine call chain 12-1..12-2
 - zones 12-4..12-6
- LightsBug** command 15-11..15-12
- Lightspeed Pascal
 - closer look at 14-1..14-7
 - compiler features 1-1..1-2
 - editing in 4-1..4-9
 - exiting 3-18
 - hardware 2-1..2-3
 - installation 2-1..2-3
 - integration of 1-3..1-4
 - introduction to 1-1..1-6
 - Macintosh environments in 14-1
 - predefined procedures and functions 117..138
 - running with a hard disk 2-3
 - running with one 400K disk drive 2-1..2-2
 - running with one 800K disk drive 2-3
 - running with two 400K disk drives 2-2..2-3
 - special features of 8-1
 - starting 3-2
 - user world within 14-1 -14-7
- Lightspeed Pascal program 89
- LightspeedC 1-1..1-2
- Line** procedure in QuickDraw C-40
- lines in **grafPort** C-26
- LineTo** procedure in QuickDraw C-40
- linking 6-2, 9-6
- LInLineF** procedure 136
- Lisa Pascal 8-2, B-10
- ln** function 125
- local coordinate system in **grafPort** C-24
- LocalToGlobal** procedure in QuickDraw C-45
- location of points in QuickDraw C-8
- log** functions in SANE D-26
- logarithm** functions in SANE D-26..D-27
- logical operations 132
- longint** 133
- LoWord** function 133
- LP1.System** 2-1..2-3, 3-1..3-2, 3-5
- LP2.Libraries** 2-1..2-3, 3-1..3-2, 3-5
- LP3.Utilities** 2-1..2-2

M

- MacinTalk library interface 8-4, 9-32

- Macintosh
 - character set 19, J-1
 - screen size C-12
- Macintosh character set J-1
- Macintosh Pascal 1-1..1-4, 8-2
- Macintosh Toolbox 8-1..8-2, 10-1
 - initialization 9-2
 - stopping at calls 12-8..12-9
- MacPasLib** 5-1..5-2, 9-3
- Macsbug** 2-2, 12, G-1..G-8
 - memory used G-3
 - program execution control G-4
 - set and display commands G-4
 - support procedures 138
- Macsbug** command 15-10..15-11
- MacTraps** 5-1..5-2, 8-2, 9-3
- MapPt** procedure in QuickDraw C-45
- MapRect** procedure in QuickDraw C-47
- MapRgn** procedure in QuickDraw C-57
- Maxbug (see Macsbug)
- Memory Manager interface E-34
- Menu Manager interface E-36
- menu summary 15-1..15-12
- MiniFinder 2-1
- mod** operator 43, 45, 47, 49, 89
- modem: 115
- Move** procedure in QuickDraw C-40
- MovePortTo** procedure in QuickDraw C-35
- MoveTo** procedure in QuickDraw C-40
- multi-dimensional array 37
- multiple indexes 37
- mutual recursion (see recursion)

N

- NaN error codes in SANE D-19, D-33
- nested comments B-5
- New** command 4-2, 15-2
- new** procedure 39, 119
- New Project...** command 2-2, 3-2, 12-10, 15-6
- NewFileName** function 135
- NewRgn** function in QuickDraw C-54
- next after functions in SANE D-25
- nil** 28, 39, 45
 - in unsigned constant 45
- nonrectangular regions in QuickDraw C-11
- non-textfiles 96, 101
- normalized number D-34
- Not a Number (see NaN)
- not** operator 43..44, 49
- Note** procedure 136
- null string 27

numbers 4
numeric comparisons in SANE D-17

O

ObscureCursor procedure in QuickDraw C-38
Observe command 3-10, 7-8, 15-11..15-12
Observe Window 7-3..7-5, 7-7..7-10, 15-12
 reloading with **Open...** 7-10
 Save As... 7-10
of
 in **array** type 23
 in **case** statement 65
 in **file** type 27
 in **record** type 25
 in **set** type 26
OffsetPoly procedure in QuickDraw C-63
OffsetRect procedure in QuickDraw C-47
OffsetRgn procedure in QuickDraw C-57
OldFileName function 135
omit function 129
Open... command 4-2, 7-10, 15-2
open procedure 96..98
Open Project... command 3-13, 15-6
opening files
 from a program 96..98
 from editor 4-2
 from Lightspeed Pascal 4-2
 in ResEdit I-4
OpenPicture function in QuickDraw C-61
OpenPoly function in QuickDraw C-62
OpenPort procedure in QuickDraw C-33
OpenRgn procedure in QuickDraw C-55
operands 43
operators 43, 47, B-7
 arithmetic 47
 binary 47
or operator 43, 46, 49
ord function 126
ord4 function 122
ordinal types 18..19, 21, 23, 25
 comparing 51
 identifier 17
ordinality 18
otherwise 65, A-4, B-4
outline field in **grafPort** C-23
outlining QuickDraw regions C-11
output standard file variable 96
overflow exception in SANE D-22

P

pack function 121
pack procedure 121
Package Manager interface E-38
packed file of char 102..103
packed in structured type 23
packed strings 24
 comparing 52
 types 6
 value with write 111
page procedure for textfiles 112
Page Setup... command 15-2..15-3
PaintArc procedure in QuickDraw C-54
PaintCircle procedure C-1
PaintOval procedure in QuickDraw C-50
PaintPoly procedure in QuickDraw C-64
PaintRect procedure in QuickDraw C-50
PaintRgn procedure in QuickDraw C-59
PaintRoundRect procedure in QuickDraw C-52
parameters 80-85
Paste command 7-8, 15-4..15-5
patStretch field in **grafPort** C-19
pattern transfer mode in **grafPort** C-28
patterns in QuickDraw C-15
pen mode in **grafPort** C-27
Penmode procedure in QuickDraw C-39
PenNormal procedure in QuickDraw C-39
PenPat procedure in QuickDraw C-39
PenSize procedure in QuickDraw C-39
PicComment procedure in QuickDraw C-62
picSave field in **grafPort** C-19
picture comments in QuickDraw C-30..C-31
picture frame C-30
pixels C-12
playing back a picture in QuickDraw C-31
pnLoc field in **grafPort** C-20
pnMode field in **grafPort** C-20, C-27
pnPat field in **grafPort** C-20
pnSize field in **grafPort** C-20
pnVis field in **grafPort** C-20
pointer function 123
pointers 39
 comparing 52
 pointer type variables 35
 table of operations 53
 types 17, 28
 values of 39
points in QuickDraw C-8
polyBBox field in **grafPort** C-31
polygons in QuickDraw C-31
polyPoints array in **grafPort** C-31

- polySave field in grafPort C-19
- polySize field in grafPort C-31
- portBits field in grafPort C-18
- portBits.bounds field in grafPort C-24
- portRect field in grafPort C-18
- portRect rectangles in grafPort C-24
- PortSize procedure in QuickDraw C-35
- pos function 128
- precedence rules for expressions 43..44
- precision 21
- pred function 127
- predefined patterns in QuickDraw C-15
- predefined procedures and functions 117..138, B-8
- Print...** command 15-2..15-3
- printer 115
- Printing Manager
 - interface E-39
 - library 8-4
- procedural parameters 82..85
- procedure**
 - declaration 12, 75
 - statement 60
- procedures
 - BInlineF 136
 - close 98
 - Debugger 138
 - DebugStr 138
 - delete 129
 - dispose 120
 - Generic 137
 - get 99
 - GetDrawingRect 131
 - GetTextRect 131
 - HideAll 130
 - inline 136
 - InlineP 136
 - input/output 95..116
 - insert 129
 - InvertCircle C-1
 - LInlineF 136
 - new 39, 119
 - Note 136
 - open 96..98
 - pack 121
 - page 112
 - PaintCircle C-1
 - put 100
 - read 101, 104
 - readln 107
 - ReadString 134
 - reset 97
 - rewrite 97
 - SaveDrawing 132
 - seek 96, 100
 - SetDrawingRect 131
 - SetTextRect 131
 - shape drawing C-1
 - ShowDrawing 131
 - ShowText 130
 - standard 13
 - Synch 135
 - unpack 121
 - WInlineF 136
 - write 101, 107
 - WriteDraw 133
 - writeln 22, 111
- program
 - activating block 13
 - execution control in Lightspeed Pascal 3-1..6-3, 6-1..6-3, 7-2..7-8
 - execution control in Macsbug G-4
 - halting execution 6-4
 - parameters 89, B-8
 - program** word-symbol 89
 - resuming execution 6-5
 - saving changes to 6-4
 - termination in Macsbug G-6
- program** syntax 89
- project
 - closing 3-12
 - compilation boxes 3-13
 - compiling 3-6, 3-13
 - compressed 9-5
 - creating 3-1..3-2
 - debugging 3-6..3-18
 - environment settings 10-2
 - error messages 3-15
 - modifying run-time 3-11, 3-16
 - observing value of variable 3-10
 - opening 3-13
 - re-running 3-6
 - resetting 3-14
 - running 3-4, 3-13
 - saving an application 3-17..3-18
 - stepping through 3-7..3-8
 - stopping at specific point 3-8..3-10
- Project** command 15-11
- Project document 1-2, 3-1
- Project Menu**
 - Add File... command 3-3, 5-1, 12-10, 15-6..15-7
 - Add Window command 4-1, 5-1, 15-6..15-7
 - Build & Save As... command 3-17, 7-2, 9-1, 9-3, 9-5, 10-2, 15-6..15-7
 - Close Project command 2-2, 3-12, 15-2, 15-6
 - New Project... command 2-2, 3-2, 12-10, 15-6
 - Open Project... command 3-13, 15-6

- Remove command 15-6..15-7
- Run Options... command 6-5..6-6, 6-5..6-6,
10-1..10-2, 10-4, 14-1, 10-1..10-2, 10-4, 14-1,
15-6..15-8, 15-6..15-8
- Source Options... command 4-5..4-6, 15-6,
15-8
- View Options... command 5-6..5-8, 15-6..15-7
- Project Window 3-1, 3-13, 5-1..5-8, 6-3, 7-4,
8-3..8-4, 15-11, B-5
 - customizing 5-6..5-7
- Date Saved column 5-7
- File column 3-3..3-4, 5-2
- MacPasLib 3-3..3-5
- MacTraps 3-3
- Options column 3-3..3-4
- Options (Compile) 3-3..3-4, 5-2, 5-4..5-5
- Selecting 3-7
- Size column 3-3..3-4, 3-6, 5-2
- Unit Name column 5-7
- Volume column 5-7
- Pt2Rect procedure in QuickDraw C-48
- PtInRect function in QuickDraw C-48
- PtInRgn function in QuickDraw C-58
- PtToAngle procedure in QuickDraw C-48
- Pull Stops command 3-11, 7-7, 15-10
- put procedure 100

Q

qualifiers 36

QuickDraw C-1

- AddPt procedure C-44
- BackColor procedure C-43
- BackPat procedure C-36
- capabilities C-6
- CharWidth function C-42
- ClipRect procedure C-36
- ClosePicture procedure C-62
- ClosePoly procedure C-63
- ClosePort procedure C-34
- CloseRgn procedure C-56
- CopyBits procedure C-60
- CopyRgn procedure C-55
- data types C-6
- DiffRgn procedure C-57
- DisposeRgn procedure C-55
- DrawChar procedure C-42
- DrawLine procedure C-1
- DrawPicture procedure C-62
- DrawString procedure C-42
- DrawText procedure C-42
- EmptyRect function C-49
- EmptyRgn function C-58
- EqualPt function C-44

- EqualRect function C-49
- EqualRgn function C-58
- EraseArc procedure C-54
- EraseOval procedure C-51
- ErasePoly procedure C-64
- EraseRect procedure C-50
- EraseRgn procedure C-59
- EraseRoundRect procedure C-52
- FillArc procedure C-54
- Filloval procedure C-51
- FillPoly procedure C-65
- FillRect procedure C-50
- FillRgn procedure C-59
- FillRoundRect procedure C-53
- ForeColor procedure C-43
- FrameArc procedure C-53
- FrameOval procedure C-50
- FramePoly procedure C-64
- FrameRect procedure C-49
- FrameRgn procedure C-58
- FrameRoundRect procedure C-50
- GetClip procedure C-36
- GetFontInfo procedure C-43
- GetPen procedure C-38
- GetPenState procedure C-38
- GetPixel function C-65
- GetPort procedure C-34
- GlobalToLocal procedure C-45
- GrafDevice procedure C-35
- HideCursor procedure C-37
- HidePen procedure C-38
- InitCursor procedure C-37
- InitGraf procedure C-33
- InitPort procedure C-34
- InsetRect procedure C-47
- InsetRgn procedure C-57
- interface E-44
- InvertArc procedure C-54
- InvertOval procedure C-51
- InvertPoly procedure C-65
- InvertRect procedure C-50
- InvertRgn procedure C-59
- InvertRoundRect procedure C-52
- KillPicture procedure C-62
- KillPoly procedure C-63
- Line procedure C-40
- LineTo procedure C-40
- LocalToGlobal procedure C-45
- MapPt procedure C-45
- MapRect procedure C-47
- MapRgn procedure C-57
- Move procedure C-40
- MovePortTo procedure C-35
- MoveTo procedure C-40
- NewRgn function C-54

ObscureCursor procedure C-38
OffsetPoly procedure C-63
OffsetRect procedure C-47
OffsetRgn procedure C-57
OpenPicture function C-61
OpenPoly function C-62
OpenPort procedure C-33
OpenRgn procedure C-55
PaintArc procedure C-54
PaintOval procedure C-50
PaintPoly procedure C-64
PaintRect procedure C-50
PaintRgn procedure C-59
PaintRoundRect procedure C-52
Penmode procedure C-39
PenNormal procedure C-39
PenPat procedure C-39
PenSize procedure C-39
PicComment procedure C-62
PortSize procedure C-35
Pt2Rect procedure C-48
PtInRect function C-48
PtInRgn function C-58
PtToAngle procedure C-48
Random function C-65
RectInRgn function C-58
RectRgn procedure C-55
 routines C-32..C-69
ScalePt procedure C-44
ScrollRect procedure C-59
SectRect function C-48
SectRgn procedure C-57
SetClip procedure C-36
SetCursor procedure C-37
SetEmptyRgn procedure C-55
SetOrigin procedure C-36
SetPenState procedure C-39
SetPort procedure C-34
SetPortBits procedure C-35
SetPt procedure C-44
SetRect procedure C-47
SetRectRgn procedure C-55
SetStdProcs procedure C-66
ShowCursor procedure C-37
ShowPen procedure C-38
SpaceExtra procedure C-41
StdArc procedure C-68
StdBits procedure C-68
StdComment procedure C-68
StdGetPic procedure C-69
StdLine procedure C-67
StdOval procedure C-67
StdPoly procedure C-68
StdPutPic procedure C-69
StdRect procedure C-67

StdRgn procedure C-68
StdRRect procedure C-67
StdText procedure C-67
StdTxMeas function C-68
StringWidth function C-42
StuffHex procedure C-65
SubPt procedure C-44
TextFace procedure C-41
TextMode procedure C-41
TextSize procedure C-41
TextWidth function C-43
UnionRect procedure C-48
UnionRgn procedure C-57
 using C-6
XorRgn procedure C-58
 Quiet NaN D-34
 Quit command 3-18, 15-2, 15-4

R

random file access 96
Random function in QuickDraw C-65
 random number generator in SANE D-30
 range of real type 21
read procedure for non-textfiles 101
read procedure for textfiles 104
readln procedure for textfiles 107
ReadString procedure 134
 real types 21
 comparing 51
 identifier 17..18
 operands 22
 SANE D-1
 with read 105
 with write 109..111
record type 24..25
 record variable 38
 records 24, 38
 with variants 26
 rectangles in QuickDraw C-8, C-11
RectInRgn function in QuickDraw C-58
RectRgn procedure in QuickDraw C-55
 recursion 13, 75..76, 80
 redeclaration
 of standard constant identifiers 20
 of standard enumerated constant identifiers 20
 of standard type identifiers 20
 within a block 13
 redefining local coordinates in **grafPort** C-24
 regions in QuickDraw C-10

- register saving conventions 11-8
- .Rel Converter 11-8
- relational operators 50-51
- remainder functions in SANE D-10
- Remove** command 15-6..15-7
- repeat** statement 66
- Replace** command 4-1, 4-9, 15-4..15-5
- ResEdit 10-4, I-1..I-10
- reserved words (see word-symbols)
- Reset** command 3-15, 6-5, 7-8, 12-11, 15-8..15-9
- reset** procedure 97
- resource declarations in RMaker H-5..H-9
- resource files 10-2, H-1..H-9, I-1..I-10
 - combining 10-2
- Resource Manager interface E-52
- resource type menu in ResEdit I-9
- restarting a halted program 7-7
- results
 - of boolean operation 49
 - of set operations 50
- Revert** command 15-2..15-3
- rewrite** procedure 97
- rgnSave** field in **grafPort** C-19
- RMaker 10-4, H-1..H-9
- round** function 122
- rounding D-11, D-21
 - direction D-21, D-34
 - IEEE D-11
 - precision D-21
 - with SANE D-11
- row width in QuickDraw C-12
- rowBytes** field in a bitmap C-14
- Run** Menu
 - 10 command 3-15
 - Auto-Save** command 6-4, 15-8..15-9
 - Build** command 2-2, 6-4, 15-8
 - Check** command 6-4, 15-8
 - Check Link** command 6-4, 15-8..15-9
 - Confirm Saves** command 6-4, 15-8..15-9
 - Don't Save** command 6-4, 15-8..15-9
 - Go** command 3-4, 3-6, 3-9..3-11, 3-13, 3-15..3-17, 6-1, 6-5, 7-3, 7-5, 7-7..7-8, 12-10, 15-8..15-10
 - Go-Go** command 3-11, 6-1, 7-2..7-3, 7-5, 7-7..7-8, 15-8..15-11
 - Reset** command 6-5, 7-8, 12-11, 15-8..15-9
 - Step** command 3-7, 3-11, 6-1, 7-2..7-8, 12-10, 15-8..15-11
 - Trace** command 3-8, 3-10..3-11, 6-1, 7-2, 7-5, 7-7..7-8, 15-8..15-11
- Run Options...** command 6-5..6-6, 10-1..10-2, 10-4, 14-1, 15-6..15-8
- run time environment B-10

- running a program 6-1..6-6
 - one step at a time 3-7, 7-4
- run-time architecture components
 - A5 world 11-2
 - heap 11-2
 - stack 11-1

S

- SANE 8
 - arithmetic operations D-9..D-10
 - class inquiries D-20
 - data types D-5..D-9
 - environmental settings D-23..D-25
 - interface E-54
 - Library 8-4
 - NaN error codes D-19
 - remainder functions D-10
 - rounding D-11
 - sign inquiries D-20
 - single type D-7
 - string conversions D-13
- Save a Copy As...** command 4-1, 4-4, 15-2..15-3
- Save As...** command 4-1, 4-4, 7-10, 15-2..15-3
- Save** command 4-4, 15-2..15-3
- SaveDrawing** procedure 132
- saving drawings in QuickDraw C-29
- saving files as entire document 7-7
- ScalePt** procedure in QuickDraw C-44
- scaling pictures in QuickDraw C-30
- scientific calculations with SANE D-5
- scope (rules of) 12..14
- Scrap Manager interface E-57
- ScrollRect** procedure in QuickDraw C-59
- search (see Find)
- SectRect** function in QuickDraw C-48
- SectRgn** procedure in QuickDraw C-57
- seek** procedure 96, 100
- Segment Loader interface E-58
- segmentation 5-3..5-4, B-10
- Select All** command 15-4..15-5
- separators 3
- sequential file access 96
- Serial Driver interface E-59
- set and display commands in Macsbug G-4
- set** type 26
- SetClip** procedure in QuickDraw C-36
- SetCursor** procedure in QuickDraw C-37
- SetDrawingRect** procedure 131
- SetEmptyRgn** procedure in QuickDraw C-55

SetOrigin procedure in QuickDraw C-36
SetPenState procedure in QuickDraw C-39
SetPort procedure in QuickDraw C-34
SetPortBits procedure in QuickDraw C-35
SetPt procedure in QuickDraw C-44
SetRect procedure in QuickDraw C-47
SetRectRgn procedure in QuickDraw C-55
sets
 comparing 52
 constructors 45, 55..56
 membership 52
 operators 50
 results of operations 50
 types 26
SetStdProcs procedure in QuickDraw C-66
SetTextRect procedure 131
shadow field in **grafPort** C-23
shape drawing procedures C-1
Show Error command 4-1, 4-7, 15-4..15-5
Show Finger command 7-4, 7-7, 15-10
Show Selection command 4-1, 4-7, 15-4..15-5
ShowCursor procedure in QuickDraw C-37
ShowDrawing procedure 131
ShowPen procedure in QuickDraw C-38
ShowText procedure 130
sign
 inquiries D-20
 manipulation D-25
 sign bit D-34
signaling NaN D-34
significand D-34
simple expression 45-46
simple statements 59
simple type identifier 17
simple types 18
sin function 125
single type D-5..D-6, D-34
size
 of object code 3-3..3-4, 3-6, 5-2
 of source files B-3
 static attribute 27
sizeof function 132
solid shapes in **grafPort** C-27
Sound Driver interface E-61
Source Options... command 4-5..4-6, 15-6, 15-8
source transfer mode in **grafPort** C-28
SpaceExtra procedure in QuickDraw C-41
special symbols 3
spExtra field in **grafPort** C-21, C-23
sqr function 124
sqrt function 124
srcBic field in **grafPort** C-23
srcOr field in **grafPort** C-23
srcXor field in **grafPort** C-23
stack 11-1
Standard Apple Numeric Environment (see "SANE")
standard constants 13
 redeclaration of 20
Standard File Package interface E-63
standard file variable
 input 96
 output 96
standard functions 13, 18
standard identifiers 18
standard ordinal types 19
standard real types 21..22
standard types 13, 20
standard units B-8
statements 11 ..12, 59
 assignment 59..60
 compound 63
 conditional 63
 (see individual word-symbols)
 simple 59
 structured 63
static size attribute 27
StdArc procedure in QuickDraw C-68
StdBits procedure in QuickDraw C-68
StdComment procedure in QuickDraw C-68
StdGetPic procedure in QuickDraw C-69
StdLine procedure in QuickDraw C-67
StdOval procedure in QuickDraw C-67
StdPoly procedure in QuickDraw C-68
StdPutPic procedure in QuickDraw C-69
StdRect procedure in QuickDraw C-67
StdRgn procedure in QuickDraw C-68
StdRRect procedure in QuickDraw C-67
StdText procedure in QuickDraw C-67
StdTxMeas function in QuickDraw C-68
Step command 3-7, 3-11, 6-1, 7-2..7-8, 12-10, 15-8..15-11
Step Into Calls command 7-4, 15-10..15-11
stepping 7-4
stop signs 3-8, 7-5..7-6, 12-11, 15-10
Stops In command 3-8, 3-11, 7-5..7-7, 12-11, 15-10
storing a bit image C-12
storing coordinates of a point in QuickDraw C-8
storing QuickDraw regions C-11
string type 27, A-4, B-4
StringOf function 134

- strings 6, 24, 27..28, 37
 - comparing 52
 - length attribute 6
 - type identifier 17
 - type variable 35
 - with read 106
 - with write 109
- StringWidth** function in QuickDraw C-42
- structured statements 63
- structured types 17, 23, 26
- StuffHex** field in **grafPort** C-21
- StuffHex** procedure in QuickDraw C-65
- SubPt** procedure in QuickDraw C-44
- subrange types 19, 21
- subroutine
 - call chain 12-1..12-4
 - name 12-3
 - variables 12-3
- subset of character set 3
- succ** function 127
- Synch** procedure 135
- System 2-1..2-3
- System Error Handler interface E-65

T

- tag field 25
- tag type 25
- testing set membership 52
- Text** command 6-1, 15-11..15-12
- Text Window 6-1, 15-12
- TextEdit interface E-66
- TextFace** procedure in QuickDraw C-41
- textfiles 96
 - standard procedures and functions 101
- TextMode** procedure in QuickDraw C-41
- TextSize** procedure in QuickDraw C-41
- TextWidth** function in QuickDraw C-43
- then** (see **if** statement)
- Three-Dimensional Graphics interface E-68
- to nearest rounding direction in SANE D-21
- to** (see **for** statement)
- tokens (Pascal) 3
- toward zero rounding direction in SANE D-21
- Trace** command 3-8, 3-10..3-11, 6-1, 7-2, 7-5, 7-7..7-8, 15-8..15-11
- Transfer...** command 15-2, 15-4
- transfer mode in **grafPort** C-28
- traps 14-7
- trigonometric functions in SANE D-29..D-30
- trunc** function 122
- txFace** field in **grafPort** C-21..C-22

- txFont** field in **grafPort** C-21
- txMode** field in **grafPort** C-21, C-27
- txSize** field in **grafPort** C-21, C-23
- type** declaration 12
- types
 - array 11-4, 23
 - booleans 11-3
 - cents 22
 - chars 11-3
 - coercion 26
 - compatibility 29
 - computational 22
 - declaration 17
 - enumerated 11-3, 20
 - extended 22
 - file 27
 - identity of 28
 - integer types 11-3
 - ordinal 18
 - pointer 28
 - pointers 11-3
 - real 11-3, 21
 - record 24
 - records 11-4
 - set 26
 - sets 11-4
 - standard 13
 - string 27
 - structured 23
 - subrange 21
 - type casts 56
 - type** declaration 12, 17, 31

U

- unary arithmetic operations 48
- underflow exception in SANE D-22
- underline in **grafPort** C-22
- Undo** command 4-1, 15-4
- UnionRect** procedure in QuickDraw C-48
- UnionRgn** procedure in QuickDraw C-57
- units 5-1, 8-2..8-4, 89..90
 - dependencies between 91, 91..92
 - implementation of 5-2, 8-2..8-3
 - interface to 5-2, 8-2..8-3
 - unit** word-symbol 89..91, A-4, -B-4
 - use(s) 8-3..8-4
- unpack** procedure 121
- unsigned constant 45
- until** (see **repeat** statement)
- upward rounding direction in SANE D-21
- user world
 - clean-up and disposal of 14-2..14-3
 - context switch from 14-3

- context switch to 14-3
- creation and initialization 14-1..14-2
- uses** 89..91, A-4, B-4
- utilities
 - Macsbug 2, G-1..G-8
 - .Rel Converter 2-1
 - ResEdit 2, I-1..I-10
 - RMaker 2, H-1..H-9
- Utilities (Toolbox) interface E-70

V

- value parameters 81
- var**
 - declaration 12
 - in formal parameter list 80
- variables
 - declaration 12, 35
 - dynamic 39
 - identifier 35
 - parameters 81-82
 - qualifiers 36
 - references 35..37
- variant 25
- vertical coordinates
 - in QuickDraw C-7
- Vertical Retrace Manager interface E-75
- View Options...** command 5-6..5-8, 15-6..15-7
- visRgn** field in **grafPort** C-18
- volume 115

W

- while** statement 66..67..67
- WIND editor in ResEdit I-8
- Window Manager interface E-76
- windows
 - activating 4-3
 - closing 4-4
 - Drawing 3-2, 3-4, 3-6, 3-9, 3-11..3-12, 3-14..3-15, 6-1, 9-2..9-3
 - editing 4-2
 - initialization of 9-2
 - Instant 3-6, 3-11, 3-13, 3-16, 4-2, 15-3
 - LightsBug 12-1, 12-3
 - moving 4-3
 - multiple 4-2
 - Observe 3-6, 3-10..3-11, 4-2, 15-3
 - Program Output 3-4
 - Project 4-2..4-3
 - Text 3-4, 9-2..9-3

- Windows Menu 4-3
 - Clipboard command 15-11..15-12
 - Drawing command 3-4, 6-1, 15-11..15-12
 - file windows command 15-12
 - Instant command 3-11, 3-16, 7-9, 15-11..15-12
 - LightsBug command 12-1, 15-11..15-12
 - Observe command 3-10, 7-8, 15-11..15-12
 - Project command 15-11
 - Text command 6-1, 15-11..15-12
- WinlineF** procedure 136
- with** statement 70, 70..71
- word symbols 3..4, A-4
 - and** 43, 45, 49
 - array** 23
 - begin** 63
 - case** 25, 63, 65..66
 - const** 11
 - div** 43, 45, 47..49
 - do** (see **for**, **while** and **with** statements)
 - downto** (see **for** statement)
 - else** (see **if** statement)
 - end** 24, 63, 65, 90
 - file** 26
 - for** 66..70
 - function** 12, 78..80
 - goto** 61..62
 - if** 63..65
 - implementation** 90
 - in** 43, 46, 50, 52
 - inline** 77
 - interface** 90
 - label** 11
 - mod** 43, 45, 47, 49
 - nil** 45
 - not** 43..44, 49
 - of** 23, 25..27, 65
 - or** 43, 46, 49
 - otherwise** 65
 - packed** 23
 - procedure** 12, 75, 60
 - program** 89
 - record** 24
 - repeat** 66
 - set** 26
 - specific to Lightspeed Pascal B-4
 - string** 27
 - summary of 4
 - then** (see **if** statement)
 - to** (see **for** statement)
 - type** 12
 - unit** 90..91..92
 - until** (see **repeat** statement)
 - uses** 89, 91
 - var** 12, 80

while 66..67
with 70..71
write parameters for textfiles 107..111
write procedure for non-textfiles 101
write procedure for textfiles 107
WriteDraw procedure 133
writeln procedure 22
writeln procedure for textfiles 111

X

XorRgn procedure in QuickDraw C-58

Z

zero rounding direction in SANE D-21

THINK's Plain Language License Statement

This manual and the software described in it were developed and are copyrighted by THINK Technologies, Inc. and are licensed to you on a non-exclusive basis. This manual, or the software, may not be copied in whole or in part except as follows:

- (1) You may make backup copies of the software for your use providing that they bear THINK's copyright notice.
- (2) You have the right to include object code derived from the libraries in programs that you develop using the software and you also have the right to use, distribute and license such programs to third parties without payment of any further license fees providing that THINK's copyright notice is affixed to any such products in the manner specified in the license agreement.

You may not in any event distribute any of the source files provided or licensed as part of the software.

You may use the software at any number of locations so long as there is no possibility of it being used at more than one location at a time. Multi-user licensing arrangements may be made by contacting THINK Technologies, Inc.

Warranty

Limited Warranty on Media and Manuals

If you discover physical defects in the media on which this software is distributed or in the manuals distributed with the software, THINK will replace the media or manuals at no cost to you provided that you return the defective materials along with a copy of your receipt to THINK or to an authorized THINK dealer during the 60 day period following your receipt of the software.

Limited Warranty on the Product

THINK warrants that the software will perform substantially as described in the User's Manual. If within 60 days of receiving the software you give written notification to THINK of a significant, reproducible error in the software which prevents its operation, and provide a written description of the possible problem along with a machine readable example, if appropriate, THINK will either provide you with corrective or workaround instructions, a corrected copy of the software, a correction to the User's Manual, or THINK will refund your purchase price upon return of all copies of the software and documentation together with a copy of your receipt.

EXCEPT FOR THE LIMITED WARRANTY DESCRIBED ABOVE, THERE ARE NO WARRANTIES TO YOU OR ANY OTHER PERSON OR ENTITY FOR THE PRODUCT EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ALL SUCH WARRANTIES ARE EXPRESSLY AND SPECIFICALLY DISCLAIMED. Some states do not allow the exclusion of implied warranties or limitation on how long they last, so the above exclusion and limitation may not apply to you. This limited warranty gives you specific legal rights and you also may have other rights that vary from state to state. IN NO EVENT SHALL THINK BE RESPONSIBLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR SIMILAR DAMAGES OR LOST DATA OR PROFITS TO YOU OR ANY OTHER PERSON OR ENTITY REGARDLESS OF THE LEGAL THEORY, EVEN IF WE HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Some states do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you. The warranty and remedies set forth are exclusive and in lieu of all others, oral or written, express or implied.

THINK

THINK TECHNOLOGIES, INC.
135 SOUTH ROAD
BEDFORD, MA 01730